

University of Warwick institutional repository: <http://go.warwick.ac.uk/wrap>

**A Thesis Submitted for the Degree of PhD at the University of Warwick**

<http://go.warwick.ac.uk/wrap/4405>

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.

# **An Intensional Implementation Technique for Functional Languages**

by

**Ali AG Yaghi**

**A dissertation submitted for  
the degree of  
Doctor of Philosophy.**

**Department of Computer Science  
University of Warwick  
Coventry  
United Kingdom**

**September 1984**

# Table of Contents:

	page number
<b>Chapter 0: Introduction</b>	
0.1- The von Neumann, Machine and Languages	0-1
0.2- Functional Programming Languages	0-3
0.3- Implementation Techniques for Functional Languages	0-5
0.3.1- Sequential Implementation Techniques for Functional Languages	0-8
0.3.2- The Reduction Approach	0-10
0.4- The Education Approach	0-14
0.4.1- Education is Demand Driven Dataflow	0-16
0.4.2- Education vs Reduction	0-19
0.5- Tagged Education and Lucid	0-20
0.6- This Thesis, The Intensional Approach to Implementing Functional Languages	0-23
0.6.1- The Structure of the Thesis	0-26
0.6.2- Mathematical Notation	0-32
<b>Chapter 1: The Language Lucid and its Implementations</b>	
1.0 Languages and Algebras	1-1
1.1 Lucid, the Dataflow Functional Language	1-6
1.2 The family of languages Luswim	1-10
1.2.1 The Language Iswim	1-11
1.2.1.A The Abstract Syntax of Iswim	1-12
1.2.1.B The Semantics of Iswim	1-13
1.2.1.C Examples	1-15
1.2.2 The Family of Algebras <i>Lu</i>	1-16
1.2.3 Examples	1-19

1.3 Nested Iteration in Lucid	1-21
1.4 Examples	1-26
1.5 Survey of Lucid Implementations	1-27
1.5.1 Basic Lucid	1-27
1.5.2 Clause Lucid	1-28
1.5.3 Structured Lucid	1-29
1.5.4 Book Lucid	1-30
1.5.5 Ostrum's Interpreter	1-30
1.5.6 Faustini's Interpreter	1-32
<b>Chapter 2: Intensional Logic and Intensional Algebras</b>	
2.0 Introduction, Semantics vs Pragmatics	2-1
2.1 Intensional Logic	2-3
2.1.1 The Intensional Language $L(\Sigma)$	2-6
2.1.2 Intensional Interpretation	2-6
2.1.2.i Indices and Possible Worlds	2-7
2.1.2.ii The Intensional Interpretation	
Function	2-8
2.2 Intensional Algebras	2-12
2.3 The Family of Intensional Algebras $\mathcal{IA}$	2-14
2.4 Synchronic and Asynchronic Operators	2-17
2.5 The Intension vs The Extension	2-20
2.6 Galaxies and Clusters for Intensional operators	2-23
2.7 Computability Requirements for Intensional Algebras	2-31
2.8 The Target Language DE	2-37
2.8.i The Syntax of DE	2-38
2.8.ii The Semantics of DE	2-39



**Chapter 3: Compiling Functions into Intensional Logic**

3.0	Introduction	3-1
3.1	The Abstract Syntax of Iwade	3-3
3.2	From Iswim to Iwade	3-4
3.2.1	The Renaming Rule	3-4
3.2.2	The Amalgamation Rule	3-6
3.2.3	The Liquidation Rule	3-8
3.2.4	The Calling Rule	3-10
3.2.5	The Addition and Deletion Rules	3-11
3.2.6	The Formal Parameter Approach to Globals	3-12
3.2.7	The Dilation Process	3-14
3.3	Functions without Globals	3-17
3.4	The Family of Intensional Algebras <i>FUN</i>	3-23
3.5	Recursion and Nested Function Calls	3-24
3.6	From Iwade to <i>DE</i>	3-27
3.6.1	The Translation Algorithm	3-27
3.6.2	Comments on the Translation Algorithm	3-28
3.7	Compiling Functions with Globals	3-30
3.7.1	The Language lpaddle	3-30
3.7.2	The Compilation of Nullary Globals	3-31
3.7.3	Example	3-32

**Chapter 4: The Compilation of Iswim**

4.0	Introduction	4-1
4.1	The Language Iswum	4-2
4.1.1	The Abstract Syntax of Iswum	4-3
4.2	From Iswim to Iswum	4-5
4.3	The Import Rule for Globals	4-6

4.4	What is peculiar to Structured languages	4-9
4.5	Lists with back pointers	4-12
4.6	The intensional algebra <i>Flo</i> and the language Florid	4-16
4.6.1	Intensional Operators in <i>Flo</i>	4-16
4.6.2	Compiling Nullary Globals	4-22
4.6.3	Compiling Global Functions	4-24
4.7	The Definition of the Algebra <i>Flo</i>	4-29
4.8	The Compilation of Iswum into Florid	4-31
4.8.1	The Compilation algorithm	4-33
4.8.2	Remarks on the algorithm	4-36
4.8.3	Example	4-37
 <b>Chapter 5: Rewrite Rules and Evaluation Techniques for Florid</b>		
5.0	Introduction	5-1
5.1	Rewrite Rules and Reduction Sequences	5-2
5.2	Rewrite Rules for the Algebra <i>Flo</i>	5-5
5.2.1-	The Rules of <i>Flo</i>	5-7
5.2.2-	Example	5-10
5.3	Tagged Eduction for Florid	5-15
5.4	Example	5-20
 <b>Chapter 6: The Compilation of Luswim</b>		
6.0	Introduction	6-1
6.1	<i>Flo</i> : The Algebra of Place and Time	6-1
6.2	The Target Language Fluid for Luswim	6-4
6.3	Rewrite Rules for <i>Flo</i>	6-7
6.4	Evaluating Fluid	6-11
6.4.1	The Reduction Method	6-11
6.4.2	Evaluating Fluid by Eduction	6-14

**Chapter 7: Conclusions and Further Work**

7.0 Conclusion	7-1
7.1 Implementing Currenting in Lucid	7-3
7.2 The Correctness of the Compilation Algorithm	7-10
7.3 Other Areas	7-11

**Appendix A**

**References**

## Acknowledgements

I would like to express my deep gratitude to:

My supervisor Bill Wadge for his continuous help and encouragement and for numerous discussions that provided the stimulation for this work.

Prof. David Park for his many comments and discussions.

All members of the Warwick semantics group who have created a stimulating environment in the early stages of this research. Also to all members of the Warwick Computer Science Department for the friendly and pleasant working environment. Special thanks go to Dr Bill McColl for the proof reading and the many comments on early versions of this thesis.

To the Ministry of Education of United Arab Emirates for the financial support of a major part of this research, and to the members of the Cultural Office of United Arab Emirates Embassy in London for their continuous help and support.

To the Science and Engineering Research Council of United Kingdom for the research fellowship in the academic year 1983/1984.

To all members of Department of Computer Science, University of Victoria, Canada for their help during my stay at the Department. To Prof. E.Ashcroft at SRI-California and A.Faustini at Arizona State University for their comments and discussions during my visit to the USA.

Finally, many thanks are due to all my friends for giving me the moral support when I desperately needed it.

## Declaration

The work described in this thesis, except where stated explicitly in the text, is my own original work. Furthermore, no portion of this thesis has been submitted in support of any other degree at any other university.

Ali AG Yaghi



## Dedication

To my family:

Ahmed , Aysha , A.Rahman , and Mohammad  
who were always on my side even when I was  
wrong; and to those I deeply missed while  
doing this research.



# An Intensional Implementation Technique for Functional Languages

Ali AG Yaghi  
Department of Computer Science  
University of Warwick  
Coventry CV4 7AL  
United Kingdom

## Abstract

The potential of functional programming languages has not been widely accepted yet. The reason lies in the difficulties associated with their implementation. In this dissertation we propose a new implementation technique for functional languages by compiling them into 'Intensional Logic' of R.Montague and R.Carnap. Our technique is not limited to a particular hardware or to a particular evaluation strategy; nevertheless it lends itself directly to demand-driven tagged dataflow architecture. Even though our technique can handle conventional languages as well, our main interest is exclusively with functional languages in general and with Lucid-like dataflow languages in particular.

We give a brief general account of intensional logic and then introduce the concept of intensional algebras as structures (models) for intensional logic. We, formally, show the computability requirements for such algebras.

The target language of our compilation is the family of languages **DE** (definitional equations over intensional expressions). A program in **DE** is a linear (not structured) set of non-ambiguous equations defining nullary variable symbols. One of these variable symbols should be the symbol **result**.

We introduce the compilation of Iswim (a first order variant of Landin's ISWIM) as an example of compiling functions into intensional expressions. A compilation algorithm is given.  $\text{Iswim}(A)$ , for any algebra of data types  $A$ , is compiled into  $\text{DE}(\text{Flo}(A))$  where  $\text{Flo}(A)$  is a uniquely defined intensional algebra over the tree of function calls. The approach is extended to compiling Luswim and Lucid.

We describe the demand-driven tagged dataflow (the **eduction**) approach to evaluating the intensional family of target languages **DE**. Furthermore, for each intensional algebra, we introduce a collection of rewrite rules. A justification of correctness is given. These rules are the basis for evaluating programs in the target **DE** by **reduction**.

Finally, we discuss possible refinements and extensions to our approach.

# Chapter 0

## Introduction

### 0.1- The von Neumann, Machine and Languages:

The architecture of all present day computers is, fundamentally, based on the concepts invented by von Neumann and others in the mid 40's. Since then, the development on the machine side was either a dress-up for the basic concepts of von Neumann or a development on the basic electronic components of the machine.

Over the last decade, the inherent defects of conventional programming languages, or what has been called von Neumann languages, have been both recognized and exposed by many computer scientists [Bac78, Bac81, Tur82, Hen80, Ber76, WaAs84, Wad78]. The reasons for such an aversion can be summarized by two terms *correctness* and *efficiency*.

On the one hand, high level conventional languages failed to give programmers the ability to reason about their programs in a mathematical way. This problem stems from the fact that, those languages are not formal systems, they are rather a syntactic representation of operational concepts. Hence, they lack simple axioms and rules of inference required for verification. The only method verifiers were left with was to make a correspondence between the language and a formal system, then use the axioms and the rules of inference of the latter to reason about programs written in the former (i.e the language). Predicate calculus played a significant role in this course; the books [Man74, Bak80] give a detailed explanation on this subject. On the other hand, these languages are inherently sequential and cannot be used to represent parallel activities which are becoming a necessity, especially in practical software



projects. Overall, the von Neumann languages failed to enable programmers to write efficient and reliable software.

Moreover, over the last decade new radical rivals for the von Neumann machine have emerged. These rivals, driven by the development of VLSI technology, are based upon multiprocessor architectures and provide highly parallel computing power. The most recent and most critical problem with conventional languages is that they cannot exploit such an amount of parallelism.

Whatever the architecture of one of these rivals might be, there is a need for a new class of languages to fully exploit the amount of parallelism evolved. Three main classes of languages emerged here:

- 1- Single-Assignment languages for Data-Flow machines.

[Gla78, Ack78, Den75].

- 2- Applicative (Functional) languages for Reduction machines.

[Tur81, Ber76, DaRe81].

- 3- Logic programming languages,

[Kow74].

## 0.2- Functional Programming Languages:

Functional programming languages have been proposed as one solution to the problems of von Neumann languages. The basic building block in these languages is the **function**. Combining these blocks to form complex ones is by the usual mathematical method of **function composition** rather than the sequential layout of *assignments*.

Apart from the ease of representing algorithms in a functional manner, there are very strong arguments to justify both the efficiency and the reliability of such languages:

1- On theoretical grounds, these languages are formal systems. Hence, they possess well defined syntax and clear modular semantics. Moreover, reasoning about programs written in such a language can be done using the language itself. This makes them powerful tools for program construction, transformation, and verification.

2- On practical grounds, they have no explicit control structure. A program in a functional language is declarative. It is a specification of the function to be computed rather than how to compute it imperatively. The only sequentiality appearing here is that induced by the data dependencies. Moreover, being a functional (applicative) expression, a program in these languages is dependent on (a function of) its arguments rather than the state of the machine. Thus, programming in these languages does not involve side effects. These grounds make functional (applicative) languages the proper tools for exploiting any amount of parallelism offered by the new parallel machines.

The potential of functional programming languages has not been widely accepted yet. This may manifest itself in the fact that two of these languages, namely LISP [McC60] and ISWIM [Lan66], had been invented well before some of

the von Neumann languages like Pascal and ADA; however, apart from theoretical and academic interests, they did not gain wide acceptance in practical fields as much as their opponents did. Functional languages were considered, until the mid 70s, as impractical and inefficient. This, basically, is due to two reasons.

The first is purely historic, and stems from our conception that a programming language should model the machine and be a descendant of machine code. Hence, it should be imperative and sequential. We quote J. Backus in [Bac81] "Originally these programs (imperative programs) were written in machine languages, then assembly language, then in FORTRAN, and then in a great variety of so-called higher level languages".

The second is due to the inefficiency of these languages on conventional machines. While conventional languages are models of the von Neumann machine, functional languages are strangers to their host (i.e. the machine). Thus, any implementation of a functional language on a conventional machine is bound to be less efficient than the implementation of any of its opponents on that same machine. This problem is, indeed, the hindrance in the way towards the acceptance of functional languages. In fact, a large portion of research on functional languages since the 60s has been dedicated to solving this problem.

The earliest of functional languages were LISP by McCarthy [McC60] and ISWIM by P.Landin [Lan60]. At present, we have many; for example:



- LISP-kit LISP by P.Henderson, [Hen80],
  - ML by R.Milner [Mil83],
  - SASL and KRC by D. Turner, [Tur81],
  - LUCID by W. Wadge and E. Ashcroft, [WaAs84],
  - FFP by J. Backus, [Bac78],
- ... and many others.

Even though all these languages are based on one formal system, i.e lambda calculus; each has its own merits. Not according to the facilities and constructs offered by each particular language but according to the overall project which was carried out in developing the language. Beside contributing to the understanding and acceptance of applicative (functional) languages, these projects had studied and offered a variety of implementation techniques and virtual machines for evaluating functional expressions.

### 0.3- Implementation Techniques for Functional Languages:

Let us, first, cast some light on the basic problem in implementing functional languages, and imperative languages with functions such as PASCAL and ALGOL. Consider, for example, the expression  $X + Y$ . The value of the expression in an environment,  $\varepsilon$  say, is the sum of the value of  $X$  in  $\varepsilon$  and that of  $Y$  in  $\varepsilon$ . Thus evaluating such an expression is straight forward using a simple stack-environment method. However, for an expression involving function application, say  $X + F(Y)$ , a simple stack-environment is not enough. Clearly, the value of  $X + F(Y)$  in an environment  $\varepsilon$  is dependent on the value of  $F(Y)$  in  $\varepsilon$ . However, the latter is not dependent on  $\varepsilon$  only, but on another environment; i.e the one defining the variable  $F$ , call it  $\vartheta$  for example. Hence, while the value of the actual parameter  $Y$  should be evaluated in the *calling environment*  $\varepsilon$ , the value of the expression  $F(Y)$  is to be computed in the *defining environment*

3. The value which the latter environment assigns to the formals of  $F$  should be that of  $Y$  computed above (in  $\varepsilon$ ). In conventional implementation techniques, used by PASCAL and ALGOL for example, this is done by transferring the control to the beginning of the body of the procedure (or the function) while passing the values of the actuals to the procedure (or the function) body either by name, reference, or value. The result, at the end of the procedure, will be sent to the return address specified at the procedure (or function) call. Conventionally, the most generally used mode of parameter passing is call-by-value [Nau63]. In this mode, the actual parameter is evaluated first, then substituted for every occurrence of the formal parameter in the function definition.

The basic concept in evaluating functional programs, however, is that the machine must not attempt to evaluate any expression unless it is known that the result is actually needed. Otherwise, computing resources may be wasted pursuing long (or worse, unending) computations which do not contribute to the final result. In particular, the actual parameters of a function must not be evaluated until it is certain that these values are actually needed. Thus, the conventional compiling techniques, used by PASCAL and ALGOL, are neither efficient nor correct for compiling functional languages.

The most correct, and efficient, mode of parameter-passing in functional languages is **call-by-need** which was first suggested by C.Wadsworth. Wadsworth in [Wad71] writes

"the call-by-value is set up at call-time but its application is delayed until execution reaches the first point at which the value is actually required. The latter view prompts the author to refer to this method as call-by-need".

In the same section of [Wad71], Wadsworth writes

"...the design of interpreters for the  $\lambda$ -calculus operating according to this strategy is one area for further research".



Henderson and Morris in [HeMo76] and, separately, Friedman and Wise in [FrWi76] were the first to study and propose such interpreters. In the first, Henderson and Morris introduced the *lazy evaluator* for evaluating pure LISP programs. The main concepts underlying this evaluator are,

perform an evaluation step only when it is necessary;

never perform the same step twice.

In [FrWi76], Friedman and Wise introduced the concept of *suspended evaluation* which is along the same lines as the mechanism of Henderson and Morris. The only addition in [FrWi76] is the introduction of hidden (from the user) parallel co-routines. Co-routines can be thought of as processes which may be suspended in the middle of computation and restarted whenever we wish.

Projects for implementing functional languages so far can be classified into two main classes. The first which is a rather conventional one is what we shall refer to here as the *sequential* (or the *state-environment*) method, for example [McC62, Lan64, Hen80]. This goes hand in hand with the von Neumann architecture by modeling the store-processor activities. The second method is the *reduction approach*, which covers almost all the recent work on implementing functional languages, for example [Berk76, Tur81, Kel79, DaRe81]. This technique can be implemented on conventional machines, but it offers a great deal of efficiency on reduction machines such as SKIM [Cla80] and ALICE [DaRe81].

In this section we shall list some of the projects in these two classes separately; and we shall describe, briefly, two main implementation techniques which represent these classes, namely Landin's SECD machine and Turner's combinatory approach.

### 0.3.1- Sequential Implementation For Functional Languages:

- McCarthy in [McC62] introduced the concept of *state vector* to describe the semantics of symbolic expressions. The *state vector* of a program at a given time is the set of current assignments of values to the variables (in scope) of the program. In other words, it is the current *environment* of the evaluation. Even though the purpose of the *state vector* was originally to offer tools for proving properties of programs, it was later developed as an implementation for the concept of environment. It is also called the *association list* [Hen80].

- Henderson in his book [Hen80] implements a purely functional variant of LISP, Lispkit LISP, using lazy evaluation [HeMo76, FrWi78] on Landin's SECD machine. The importance of Henderson's kit comes from the fact that the compiler compiles itself and any other Lispkit program. Thus offering a portable experimental functional language. Besides, the contribution of the book to a good understanding of the semantics and implementation of functional languages is widely recognized.

We give here a brief description of Landin's SECD machine as an example on this approach to implementation.

#### Landin's SECD Machine

The source language which Landin implemented in [Lan64] and developed later as the family of languages ISWIM [Lan66] is the set of AE (*applicative expressions*). These are, basically,  $\lambda$ -expressions. The semantics of the language was laid down in a modular comprehensive schema. Landin's SECD machine was a virtual machine to mechanize the evaluation of such expressions.

In the machine, both environments and expressions are represented as lists. The value of an expression  $e$  in an environment  $\vartheta$  is represented as a *closure*. A *closure* is a pair of lists; the first is the representation of the

expression and the second represents the environment. Hence, if  $\varepsilon'$  is the list representing  $\varepsilon$ , and  $\vartheta'$  is that representing  $\vartheta$ , then the value of  $\varepsilon$  at  $\vartheta$  is represented by the closure  $\langle \varepsilon', \vartheta' \rangle$

A state of the machine is a quadruple  $(S, E, C, D)$  where

$S$  is a **stack**, a list holding intermediate results during the evaluation of expressions.

$E$  is an **environment**, a list of name-value pairs.

$C$  is a **control**, a list representing the machine code of the expression.

$D$  is a **dump**, a state of the machine (a quadruple).

Each instruction of the machine corresponds to a state transition, or (machine transition [Hen81]). A *transition* from the state  $(s, e, c, d)$  to  $(s', e', c', d')$  then, can be represented by

$$(s, e, c, d) \longrightarrow (s', e', c', d')$$

The power of this machine was shown clearly in [Hen81]. P.Landin in the original work did not discuss the choice of machine language which runs the machine. Nor did he discuss the actual representation of applicative expressions in the machine. Henderson in [Hen81] represents such expressions as lists and gives a complete set of instructions as mnemonic operators.

Clearly, Landin's SECD machine is a state-environment model, hence purely sequential. It is an efficient technique for evaluating expressions on von Neumann machines; however, it cannot be used for parallel evaluation of applicative expressions. The main hindrance in implementing such a technique on a parallel machine will be the idea of carrying the *closure*, in particular the environment part, during the evaluation. It is worth mentioning here that Abramsky in [Abr82] introduced the SECD-M (Multiprogramming SECD)



machine. However, the main objective was to run his SASL-like language on a conventional machine.

### 0.3.2- The Reduction Approach:

A program in a reduction language is an applicative (a functional) expression. An applicative expression is a combination:

$\langle \text{function} \rangle \langle \text{argument} \rangle$ .

A  $\langle \text{function} \rangle$  or an  $\langle \text{argument} \rangle$  in a reduction language over an algebra of data types (for example the natural numbers represented by their usual symbols) is defined recursively as either an element in the signature of the algebra, i.e. the set of symbols (for example 0, +, 3, \*), or an expression.

Evaluating an expression by reduction is to rewrite over the expression to a (computationally) simpler form.

There are two forms of evaluation by reduction, depending on how the process of *simplification* is actually carried out. They are **string reduction** and **graph reduction**.

In **string reduction**, each occurrence of the subexpression in the main expression to be reduced accesses and manipulates a separate copy of the new form of the subexpression. For example, in evaluating the expression

$$(X-1) * (X+1)$$

where  $X = 3*Y$  we replace every occurrence of  $X$  in the expression by  $3*Y$  yielding the expression

$$((3*Y)-1) * ((3*Y)+1)$$

In other words, it is equivalent to call-by-name parameter passing. Each occurrence of  $X$  will access and manipulate a separate copy of the definition of  $X$ . This form of reduction is used in the GMD reduction machine of K.Berkling [Berk71, Berk76] and the Newcastle parallel string reduction machine of Treleaven et.al. [TrMo80].

In **graph reduction**, access to an expression is by reference, and thus arguments are shared using pointers. Clearly, this form of reduction is more efficient than string reduction because once an expression is evaluated all references to it access that value. This form of reduction is used in many reduction machines; for example

the Utah applicative multiprogramming system of R.M.Keller et.al. [Kel79] which is a parallel graph reduction machine,

the Cambridge SKIM graph reduction machine of T.J.Clarke et.al. [Cla80],

the parallel graph reduction machine ALICE of J.Darlington and M.Reeves [DaRe81]

and the reduction machine of M.Sleep and F.Burton [SlBu81].

We describe here Turner's combinatory approach to implementing functional languages by reduction as it is the basis of many present reduction machines.

### **Turner's Combinatory Approach:**

The main concept in this approach is to compile applicative (functional) expressions into combinatory expressions in which no bound occurrences of variables occur. Then such expressions, represented as a graph are evaluated using normal graph reduction.

The compilation part of the technique consists of two main steps *combination* and *abstraction*. *Combination* is to translate an applicative expression into an expression in which the only operation allowed is function application. It is based on *currying*. *Currying*, after the logician H.Curry, is a technique for reducing a first order function with many arguments to a higher

order function of one argument only. Hence, by *combination* the definition

$F\ x = E$  where  $E$  is an applicative expression

is translated into

$F\ x = E'$  where  $E'$  is the 'combined' version of  $E$

i.e the only operator in  $E'$  is function application.

*Abstraction* removes all occurrences of bound variables from the expression  $E$ . This is done by defining a set of constant symbols called *combinators*, and defining the abstraction operations using these combinators. While the abstraction operations are considered as rewrite rules for the compiler, the definitions of these combinators form rewrite rules for the evaluator. The above definition becomes

$F = E'$  where no bound variables occur in  $E'$ , and the only operation is function application.

The result of the compilation is an expression in which the only operation which occurs is function application, and without occurrences of bound variables. These expressions are called *combinations* [Tur79]. These combinations, which form the object code for the machine, are represented as a graph in which the nodes are function applications and leaves are constant symbols (including combinators). Then, the combinator definitions are implemented as rewrite rules (reduction rules) on such a graph. The evaluation of the object code is by normal graph reduction of [Wads71].

The importance of the reduction approach in general, and of Turner's work in particular, is due to many facts:

- It is a departure from the conventional state-environment model. Even though it can be implemented on a von Neumann machine (an implementation already exists [Tur81]), it offers a new mode of architecture *reduction machines*. The



object of the compilation can be a machine code for such architecture.

- It is not committed to a mono-processor-store machine. This comes from the fact that no environment (in the conventional meaning of the word) is needed as there are no bound variables. Thus, the approach can be the basis for a parallel architecture on the lines of ALICE [DaRe81].

- The use of normal graph reduction makes use of two advantageous concepts in reduction; normal order reduction (evaluate arguments only if needed) and applicative order reduction (arguments are evaluated at most once).

However, there is one main objection on this approach, and on reduction machines in general. That is, it is a *destructive* technique. Once we reduce an expression it is overwritten by the new form, and hence if a run time error occurs we cannot trace back its origin in the source program. One way out is to store all the intermediate expressions in the reduction process, but this will cause a great deal of inefficiency. However, as run time errors in applicative languages are usually caused by type clashes, an efficient way to solve this problem is to introduce strong typing to the language. Then, to do type checking at compile time [WeEl83].

#### 0.4- The Eduction Approach:

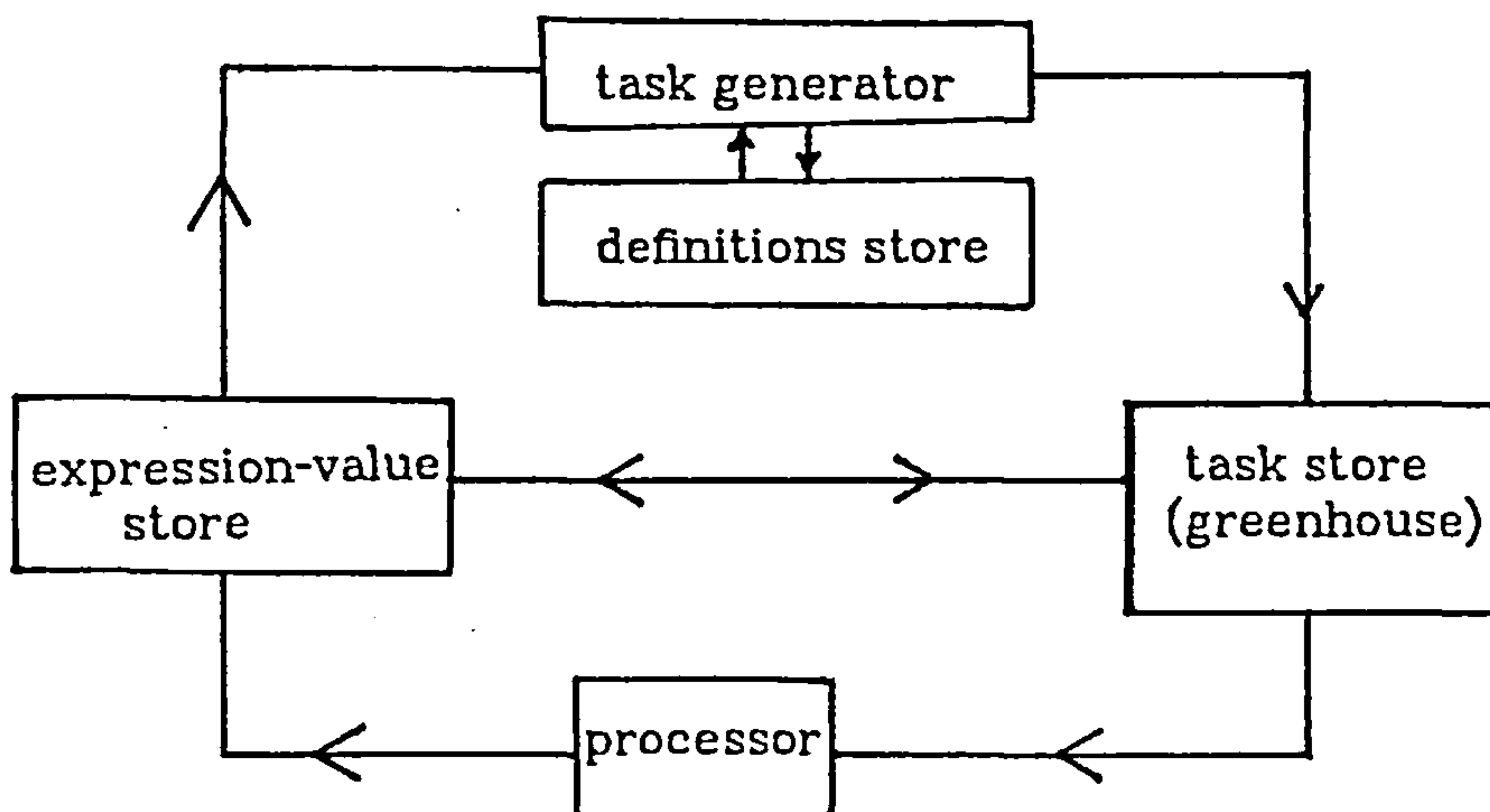
This thesis is concerned with one technique to organize the evaluation of functional languages- "eduction". The word **Eduction**, from the Oxford English Dictionary, is "The action of drawing forth, eliciting, or developing from a state of latent, rudimentary, or potential existence; the action of educating (principles, result of calculations) from the data".

Even though the word 'eduction' has been coined, by W.Wadge and E.Ashcroft, to this mode of evaluation recently [Ash84], the mode itself is not a new one. It has been used before as an implementation technique for Lucid [Carg76, Ost81]. In fact, if we think of lazy evaluation as a technique in which evaluation is controlled by demands then "eduction" is a pure form of lazy evaluation.

To explain the intuitive concept of eduction, the following scenario is suggested between the processor of an eduction machine and the other modules of the machine:

A processor is given a task, say "evaluate  $X+Y$ ". The processor responds by asking for the values of both  $X$  and  $Y$ . The other modules supply the processor with the values of  $X$  and  $Y$ , say  $x$  and  $y$ . The processor computes the sum  $x+y$ .

We give here a crude sketch of an eduction machine and explain, roughly, the eductive evaluation of a simple equational program. The reader is reminded that the description in this section is not intended to suggest a particular machine architecture but rather an intuitive description of the eduction process itself.



The machine consists of a processor together with the following modules:

- 1- The *expression-value* store: which holds the values of previously computed expressions which may be used in future computations.
- 2- The *definitions store*: which holds the definitions of the program to be evaluated.
- 3- A *task generator* which receives demands for evaluating variable symbols, fetches the definition of the symbol from the definition store and generates a task for evaluating the expression defining such a symbol.
- 3- The *task store* or the *greenhouse*: which stores the tasks awaiting ripening. A task ripens when all its operands are computed.

Consider the following program in lswim

**X+Y where**

**X = Z \* Y;**

**Y = 2 \* A;**

**Z = 3 \* A;**

**A = 4;**

**end**

The definitions in the program are loaded into the definitions store of the machine, and the computation is triggered by generating a task for evaluating



the expression  $X+Y$  (this is the value of the program or the expression according to the semantics of lswim).

The task for evaluating  $X+Y$  will wait in the greenhouse. The latter checks for the values of  $X$  and  $Y$  in the expression-value store. If either  $X$  or  $Y$ , say for example  $Y$ , has not been computed yet then a request is sent to the task generator to generate a task for evaluating  $Y$ . The task generator fetches the definition of  $Y$  from the definition store and generates a task for evaluating  $2*A$ .

This again travels to the greenhouse which demands the value of  $A$  from the expression-value store. As  $A$  has not yet been computed, a demand is sent to the task generator to create a task for evaluating  $A$ . From the definitions  $A=4$ , and the value of  $A$  is sent to the greenhouse. The task for evaluating  $2*A$  is ripened now, and a new task for evaluating  $2*4$  is sent to the processor to be computed. When it is computed, the value is sent to the expression-value store and to the greenhouse to ripen new tasks and so on.

#### 0.4.1- Education is Demand Driven Dataflow:

The basic concept underlying the "dataflow" model of computation is that it proceeds by flow of data around a *dataflow network* rather than by the conventional flow of control of a flowchart. A *dataflow network* is a directed graph in which nodes represent processing stations while arcs represent the paths which link these stations together and along which data tokens travel in the computation process. This is a very general view of *dataflow* and there are various views as to how the computation process in dataflow actually proceeds [KaMi66, Den79, WaGu82, Dav78, DML77, ArKa81, Fau83, Ash84]. Nevertheless, all these views agree on the main principle of dataflow. This is, computation proceeds *asynchronously in parallel*.

The first main aspect of distinction between the dataflow schools is whether computation is driven by the flow of data (**data-driven dataflow**), for example [Dav78, DML77, WaGu82]; or driven by the flow of demands in the network (**demand-driven dataflow**), for example [Ash84, ArKa81, Pil83]. In the first, a node *fires* (starts computing) when there is at least a *daton* (a data token) at each of its input-ports. The result, if there is to be a result, travels from the output-port of the node to the next node via the data-path connecting the two, figure 0.4.1A.

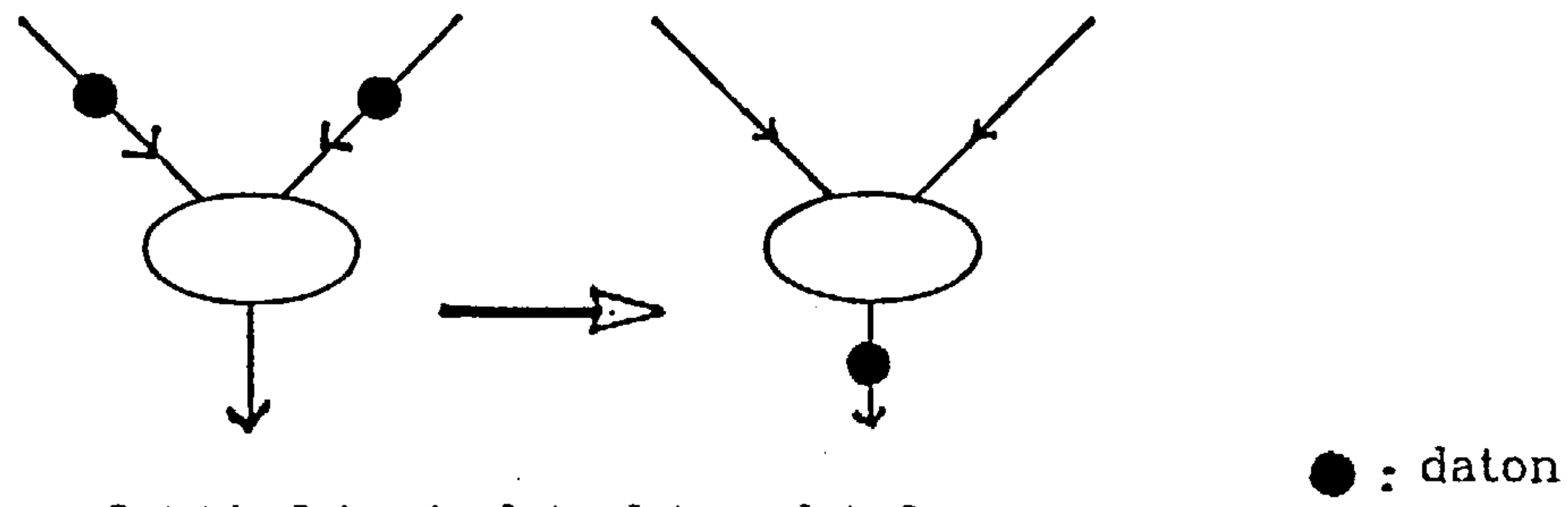


figure 0.4.1A: firing in data-driven dataflow

In demand-driven dataflow, a node fires when there is a daton at each of its input-ports, and there is a *demand* on its output-port. If there is a demand on the output port and one of the inputs does not have a daton, the node creates a demand and such a demand travels upwards via the data path to the node feeding the present one, figure 0.4.1B

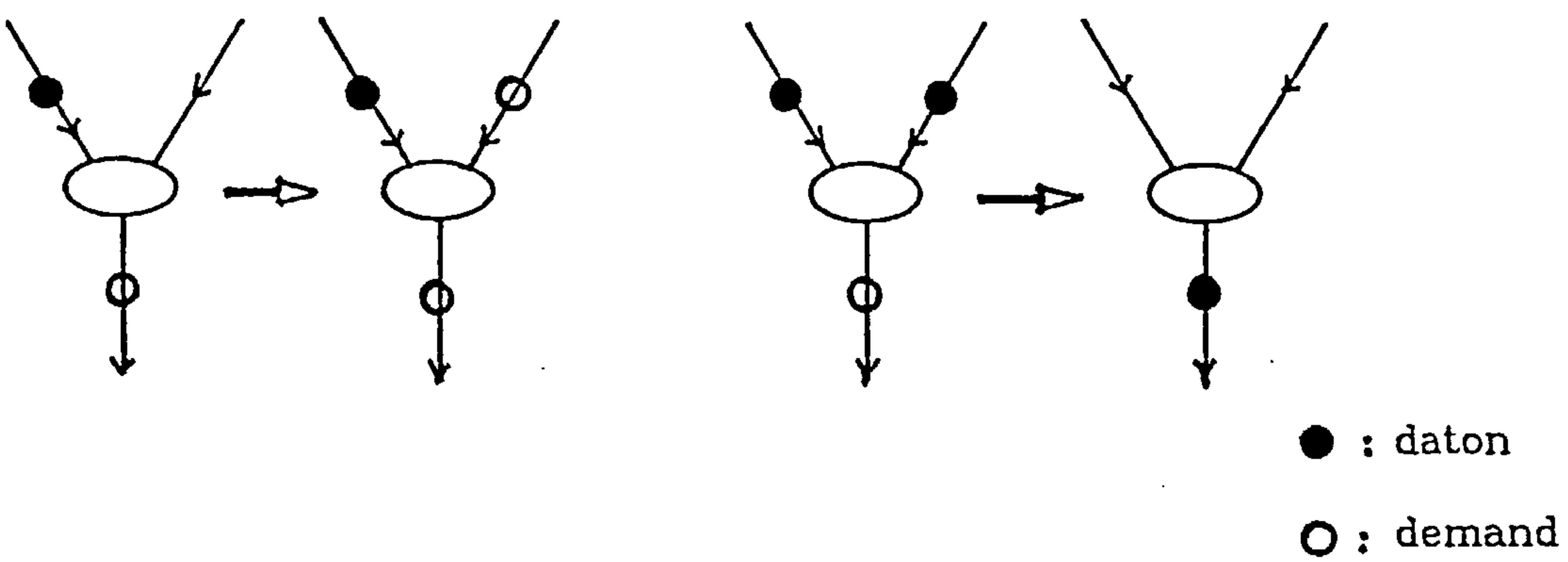


figure 0.4.1B: firing in demand-driven dataflow



The second main point of distinction between the dataflow schools is concerned with the arcs (data paths) connecting the processing stations of the network. In **pipeline dataflow** [KaMi66, Dav78, DML77, ArGo78, Fau82, Pil83], the data paths are considered as pipes through which datons travel in a FIFO manner (first-in, first-out). Thus, the paths are queues acting like buffers between the nodes, the first to be produced by a node is the first to be swallowed by the next. In **tagged dataflow** [WaGu82, ArKa81, Ash84], the data paths between the nodes indicate the destination which the output of a certain node should take, but do not impose any order on such output datons. A data token (a daton) consists of the data to be processed together with a tag carrying information about the data. For example, in the Manchester tagged dataflow machine [GuWa77, WaGu82], the tag carries the iteration level (in loops), the re-entry colour (in function calls) and the destination (the address of the node) of the daton and so on.

It is worth noting here that if the tag carries the order in which datons are produced, then tagged dataflow can simulate pipeline dataflow. We can simply impose the restriction that an importing node consumes the output of the exporting node in order.

Clearly, within such a taxonomy, the eduction approach lends itself directly to the demand-driven mode of dataflow. In fact, E.Ashcroft and W.Wadge, who first used the word 'eduction', had originally coined it to such a mode of dataflow. They argue that, if we think of the word *dataflow* as describing those systems in which a program is a net and data tokens flow between the stations of this net through its arcs then the term "demand-driven dataflow" is a contradictory one.

"Eduction is sufficiently different from normal dataflow that we thought it appropriate to give it a new name

- (1) to avoid confusion,
- (2) to emphasize that it is a new concept,
- (3) to separate it from any preexisting opinion that one has about dataflow." [Ash84]

#### 0.4.2- Education vs Reduction:

At present, the most active area of research in demand-driven evaluation of functional languages is the reduction approach. Even though many researchers, for example [TrMo80, TBH82, ArKa81], consider such an approach as demand-driven dataflow, we believe that a contrast should be made between the two. Such a contrast between reduction and education (demand-driven dataflow) can be summarized by the following:

- 1- Reduction does not inherently involve demands. It is a destructive process in which we start with an expression and continuously manipulate it until we produce its simplest form. Education, on the other hand, is a conservative process. The program is stored as read-only.
- 2- The result of demanding the value of an expression in education is always a data object (such as a numeral), while the result in reduction is an expression (a reduced form of the original one) which might be a data object.
- 3- In education we only demand, evaluate and store the expressions which occur in the program (or the original expression). In reduction however, we may demand a value of an expression which does not occur in the program; i.e. we may demand a value of an expression which is a result of a rewriting or a reduction step.
- 4- The reduction approach inherently involves manipulating hierarchical data structures and favours recursive algorithms, while education (demand-driven dataflow) favours iterative algorithms and linear data structures.

5- The eduction approach allows writing heuristic procedures for garbage collection; while in reduction such procedures may discard values which are irreplaceable.

### 0.5- Tagged Eduction and Lucid:

Denotationally, the value of a variable in Lucid is an infinite sequence. Operationally however, it is often very useful to think of the value of an expression as changing with time. Time here is not meant to be *real time*, but rather a conceptual time which enables us to talk about the first value of an expression, the second value, the third and so on. For example, the value of the expression  $X$  defined as

$$X = 0 \text{ fby } X+5$$

(fby is a Lucid operator read *followed by*) is the infinite sequence  $\langle 0, 5, 10, 20, \dots \rangle$ . Thus, operationally, the value of  $X$  is initially 0 and changes subsequently to 5, 10, 15, etc.

Clearly, for a language with a domain of infinite objects like Lucid, demanding the value of an expression will lead to a non-terminating computation. Thus, it is necessary to demand the value of an expression at a particular instant in time. In the example above, we may demand the initial value of  $X$ , the 7th or the 4th and so on.

Hence, in the sketch of the eduction machine described above, the expressions which move around the communication channels are tagged with natural numbers representing the time at which such an expression should be, or is, evaluated. Also, when the task generator generates a task for evaluating a certain expression it specifies the particular value of the expression. We call such a machine a **tagged eduction machine**.

Assume for example that a task is generated to evaluate the third value of



the expression  $X+Y$  where

$$X = 0 \text{ fby } X+5$$

$$Y = 1 \text{ fby } Y*2$$

(the operator  $+$  in Lucid can be thought of as a pointwise filter which takes two values as inputs and outputs their sum). That is, we want to evaluate the expression

$$X+Y \text{ tagged with the natural number } 2^\dagger$$

The scenario in tagged education is similar to that in normal education:

The processor is given the task of evaluating the expression  $X+Y$

at time 2. The processor demands the values of  $X$  and  $Y$  at time 2.

Given such values, the processor returns the sum.

In more detail, once the task of evaluating  $X+Y$  at time 2 is generated, it waits in the greenhouse which sends two demands to the expression-value store asking for the values of

$X$  at 2 and that of

$Y$  at 2.

Assuming that neither the value of  $X$  nor that of  $Y$  at time 2 has been evaluated, the task generator then generates two new tasks, one for each value. Let us, for example, take the task for evaluating the second value of  $X$ ; it is the task for evaluating the expression

$$0 \text{ fby } X+5 \text{ tagged with the number } 2$$

As we shall see from the semantics of Lucid (Chapter 1), the value of the above expression is the value of the sum

$$5 + (\text{the value of } X \text{ at } 1).$$

Hence, when the processor receives the task of evaluating  $X$  at 2 it demands

---

<sup>†</sup> the third value of an expression is its value at time 2 because the first natural number is 0.

the value of  $X$  at 1. A new task then is generated to compute

$0 \text{ fby } X+5$  tagged with the number 1

Similarly, this leads to a new task being generated; it is

$0 \text{ fby } X+5$  tagged with the number 0

The value of the latter is 0, and the availability of this value in the greenhouse will ripen all the previous tasks and send them to be processed. Therefore,

$X$  at 0 is 5

$X$  at 1 is  $5 + (X \text{ at } 0) = 10$

$X$  at 2 is  $5 + (X \text{ at } 1) = 15$

The procedure we used to evaluate the expression above was recursive. We started by demanding the value of  $X$  at 2 then recurring back until we demanded the value of  $X$  at 0. This is not a necessary condition for evaluating  $X$  at 2. Our purpose was to explain the process of education only. We can evaluate  $X$  at 2 iteratively by generating the value of  $X$  at a subset of the natural numbers, then let the education process consume what it needs from such a collection of values. The latter approach (combining dataflow with education) has been termed **eazyflow** [JaAs84] and is the basic principle of the tagged education engine of E.Ashcroft [Ash84].

It is not always the case that a demand for an expression at a certain instant in time will result in demanding its argument at that same instant (We shall call such operators **synchronic**; see Chapter 2). For example, a demand for the value of the expression

$\text{first}(X)$  at time 20

(**first** is a Lucid operator, it returns the initial value of its argument) will result in a demand for the value of  $X$  at time 0.

## 0.6- This Thesis, The Intensional Approach to Implementing Functional Languages

In this thesis we present a new implementation technique for functional languages based on intensional logic of Montague [Mon74, Car49]. Our approach is not committed to any particular hardware, or to any particular evaluation technique. Nevertheless it lends itself directly to demand driven tagged dataflow architecture; i.e. to tagged education. Such an approach offers an efficient technique to implementing functional languages in general and dataflow languages (such as Lucid) in particular.

Intensional logic is the study of *indexical expressions*, that is "words and sentences of which the reference cannot be determined without the knowledge of the context of use" [Mon74]. Thus, if  $U$  is a collection of contexts and  $D$  is a domain then the *intensional value* of an expression is a family of indexed objects. Each of these objects denotes the *extensional* value which the expression takes at that particular index. That is, the intensional value of an expression is a function which maps the set of contexts  $U$  to the domain of discourse  $D$ . We call the set of contexts the universe, or the set, of possible worlds.

For example, if we consider the universe of possible worlds to be the set  $\omega$  representing years, then the value of the proposition represented by the sentence

"John is a student"

is a function which maps each year (world in the universe  $\omega$ ) to a truth value (an element in the domain  $T$ ). That is, it is a function in  $[\omega \rightarrow T]$ , where  $[\omega \rightarrow T]$  denotes the set of all functions from  $\omega$  to  $T$ .

Based on *intensional logic* is the concept of *intensional algebras*. We define an intensional algebra over a signature  $\Sigma$  (a collection of constant



symbols) to be a triple  $\langle U, F, D \rangle$  where  $U$  is a universe of possible worlds,  $D$  is a domain of objects, and  $F$  is the intensional interpretation function (see Chapter 2). The function  $F$  assigns to every  $n$ -ary symbol in the signature of the algebra a function of degree  $n$  over the set of possible intensions. The set of possible intensions is the set of all functions from the universe  $U$  to the domain  $D$ . Thus, the intensional value of an  $n$ -ary symbol in the signature is a function in  $[[U \rightarrow D]]^n \rightarrow [U \rightarrow D]$ . We call  $D$  the **extensional domain** of the algebra, and  $[U \rightarrow D]$  the **intensional domain**.

The basic idea in this approach is to translate functional languages, such as Iswim†, into nullary order equational languages (equational languages without functions) over intensional algebras. The family of target languages we use in this thesis is called *DE* (Definitional Equations) and is defined in Chapter 2. The complexities associated with the functional language, such as calling and binding, are transformed then into intensional operators in the target algebra.

For example, the function definition

$$F(X) = X * X$$

is translated into the nullary 'intensional' equation

$$F = X * X$$

Knowing that the value of the function  $F$  depends on the value which its formals take from one world to another, we propose the set of function calls as the universe of possible worlds. Thus the equation

$$F = X * X$$

states that, in any world in the universe, the value of  $F$  equals the value of the expression  $X * X$ . We introduce the intensional operator symbol  $\text{act}$  where for a formal  $Z$  and expressions  $a, b, c, \dots$

---

† We use the word Iswim to refer to the language defined by E.Ashcroft and W.Wadge in [AsWa80, WaAs84], and the word ISWIM (If You See What I Mean) to refer to the language defined by Landin in [Lan66].

$$Z = \text{act} ( a , b , c , \dots )$$

means, informally, that the value which the formal  $Z$  takes is,  $a$  at the first call,  $b$  at the second call,  $c$  at the third, and so on. The value of  $F$ , then, at the world (or context)  $i$  will be the value of  $F$  at the world in which the formal  $X$  takes the value of the actual parameter of the  $i$ 'th call of  $F$  in the source program. Moreover, we introduce the family of operators  $\text{call}$ , where  $\text{call}_i F$  denotes the  $i$ 'th call of the non-nullary variable symbol  $F$ . Hence the equations

$$F(X) = X * X$$

$$Y = F(2) + F(3)$$

are translated into

$$F = X * X$$

$$Y = \text{call}_0 F + \text{call}_1 F$$

$$X = \text{act} (2,3)$$

The value of  $\text{call}_0 F$  is the value of the nullary symbol  $F$  at the world 0, which is the value of  $X * X$  at 0. The value of  $X$  at 0 is the value of  $\text{act}(2,3)$  at 0 which is 2.

In section 0.7, we have demonstrated the process of tagged eduction using the language Basic Lucid (the subset of Lucid which does not allow function definitions). An expression in such a process is tagged with an instant in time which specifies the particular member in the value sequence of the expression; i.e. a tagged expression in the eduction of Basic Lucid looks like

expression	tag : a point in time
------------	-----------------------

For example, the value of the tagged expression

0 fby X+4	20
-----------	----



is the 20th element in the value sequence of the expression  $0 \text{ fby } X+4$ .

In functional languages, such as ISWIM and Iswim, the expression in the process of tagged eduction should carry a tag which specifies the place in the tree of function calls. That is, a tagged expression in the eduction of a functional language will consist of

expression	tag: a place in the tree of function calls
------------	--

For example, a tagged expression in the eduction of a functional language will look like

$\text{call}_5 \text{ F} + 10$	a place in the universe
--------------------------------	-------------------------

The complexity of the universe of places is, as we shall see, dependent on the particular language to be implemented. For example, the universe of an intensional model (intensional algebra) of a non-structured functional language is the set of lists of natural numbers. The head of each list represents the present function call while the tail represents the subsequent calls which lead to the present one.

0.6.1- The Structure of the Thesis:

This thesis has two interrelated objectives:

- (1) To propose a new implementation technique for functional languages based on intensional logic and tagged eduction
- (2) To implement the language Lucid [WaAs84] which is a family of functional dataflow languages.

Apart from our own interest in Lucid, the idea of using such a language as the case study for the proposed implementation technique comes from the following facts:

- It is an eductive language; i.e. it lends itself directly to tagged lazy evaluation. All existing implementations of Lucid are lazy and cannot be otherwise.

- It is a functional language. Thus using it as an example in the thesis will cast light on implementing this class of languages in general.

- It is also a dataflow language. Programs in Lucid can be easily mapped to dataflow nets. Operators in Lucid can be visualized as continuously operating filters.

- Basic Lucid which is the nucleus of Lucid is an intensional language. The universe of possible worlds is the set of natural numbers, and the value of an expression is a sequence representing its value at each point in the universe. In other words, it is a function from the set of natural numbers to the domain of computation. The reader should also note that the Lucid operators are 'intensional' because they facilitate switching between the contexts of evaluation or the worlds of the universe. For example, the value of the expression  $\text{first}(X)$  at any world in the universe (the set of natural numbers) is the value of the expression  $X$  at the world represented by the number 0.

In Chapter 1, we give some preliminary concepts related to this thesis. We describe the idea of viewing a language as a family in which each member is uniquely determined by an algebra of data types. This view of languages was first introduced by Landin [Land66], then carried out formally in the design of Lucid by E.Ashcroft and W.Wadge [AsWa79, AsWa80, WaAs84]. Such a view is adopted throughout this thesis. An algebraic construction of the family Lucid, following the approach of E. Ashcroft and W.Wadge, will be described. In such a construction, the language Lucid (Structured Lucid or Full Lucid) is the language  $\text{Iswim}(\mathcal{L}\mathcal{U})$ . The language Iswim, developed by E.Ashcroft and W.Wadge, is a variant of Landin's ISWIM. The algebra  $\mathcal{L}\mathcal{U}(A)$ , for any algebra of data types

$A$ , is the algebra of sequences of elements of the type  $A$  with the Lucid operators (*first*, *next*, *fby*, ...). Therefore, while the algebra *L<sub>u</sub>* facilitates computing with sequences, *lswim* facilitates programming in a structured functional manner.

In Chapter 2, we describe the basic concepts in intensional logic. We define the family  $L$  of intensional languages, then introduce the concepts of intensional structure and intensional environment.

Based on this logic, we define an intensional algebra to be an intensional structure (logically). It is a triple  $\langle U, F, D \rangle$  where  $U$  is a universe of possible worlds,  $D$  is a domain of objects, and  $F$  is the intensional interpretation function which assigns meaning to different symbols of the signature. The intensional value of an  $n$ -ary symbol in the signature is a function in  $[(^U D)^n \rightarrow ^U D]$ . We call  $D$  the *extensional domain* of the algebra, and  $^U D$  the *intensional domain*. We discuss then the computability requirements for intensional algebras within the context of eduction. We show that an intensional expression is educable (can be evaluated by eduction) if the information required in the evaluation process is finite.

In Chapter 3, we describe the main concept in our technique; i.e. the compilation of functions. We introduce the compilation of the non-structured functional language *lwade*. *lwade* is the subfamily of *lswim* which

- does not allow the *where*-clause expression except as the outermost casing of a program, and
- does not allow the occurrences of nullary global symbols in function definitions.

We define the family of intensional algebras  $FUN$ ; the member  $lwade(A)$ , for an algebra of data types  $A$ , is compiled into  $DE(FUN(A))$ . That is, a program in  $lwade(A)$  will be compiled into a set of nullary definitions (equations) over the



intensional expressions of the algebra  $FUN(A)$ . The universe of  $FUN$  is the set of lists of natural numbers; the head of each list represents the present function call while the tail represents the sequence of calls which lead to the present one.

The algebra  $FUN$  and the technique described in Chapter 3 is sufficient to compile and implement the whole family *Is<sub>wim</sub>*. We can transform programs in *Is<sub>wim</sub>* into equivalent *Is<sub>wade</sub>* programs using a collection of program manipulation rules then use the technique to compile such programs into  $DE(FUN)$ . The procedure of transforming *Is<sub>wim</sub>* programs into equivalent programs in *Is<sub>wade</sub>* is described.

At the end of the Chapter, we show that, by enriching the algebra  $FUN$  with the operator symbol  $\gamma$ , the technique can handle occurrences of nullary global symbols in function definitions.

In Chapter 4, we take a step forward and introduce the family of functional structured languages *Is<sub>wum</sub>*. The only difference between *Is<sub>wim</sub>* and *Is<sub>wum</sub>* is that, a variable symbol in the latter is defined (either occurs as a left hand side of an equation or as a formal in a function definition) at most once in the whole program. Therefore, transforming *Is<sub>wim</sub>* programs into *Is<sub>wum</sub>* is straight forward using the renaming rules described in Chapter 3.

We show, by means of examples, that the set of lists of natural numbers is not sufficient as a universe for the intensional algebra in the compilation of structured functional languages. We need a richer algebra than  $FUN$ ; in particular, we need to know the place where a function is defined so that we bind its globals to their definitions properly.

We define the intensional algebra  $Flo$  whose universe is a special class of lists. We call such lists **b-lists** or **lists with back pointers**. A b-list has a head and two tails, a **dynamic tail** which represents the sequence of function calls



which lead to the present one (represented by the head of the b-list) and a **static tail** which represents the place where the function (being invoked) is defined. *Flo* has a collection of intensional operators which resolve the complexities associated with structured functional language such as binding globals and calling actuals.

Given an algebra of data types  $A$  then, the language  $\text{Iswum}(A)$  is compiled into the member  $DE(Flo(A))$  of the family  $DE$ . We call the family of target languages  $DE(Flo)$  the family **Florid**.

The evaluation of our target family **Florid** is discussed in Chapter 5. We describe two approaches for evaluating programs in **Florid**. The first is **reductive** and is based on a set of rewrite rules for the intensional algebra *Flo*. A collection of rewrite rules for *Flo* are given and are justified using the definition of *Flo*. The second evaluation approach is the **tagged eduction** and is described along with the lines of tagged eduction described before.

If we call the algebra *Lul* as the algebra of time because its universe is the set of natural numbers, and the algebra *Flo* as that of place because its universe is the set of b-lists representing points in the set of function calls; then the algebra resulting from combining *Lul* with *Flo* may be called the algebra of place and time. This is the subject of Chapter 6.

In this Chapter, we introduce the family of intensional algebras *Flu* as the target family for the compilation process of *Luswim*. That is, given an algebra of data type  $A$ , the language  $\text{Luswim}(A)$  is compiled into the intensional equational language  $DE(Flu(A))$ . The family of languages  $DE(Flu)$  is called **Fluid**.

Given an extensional algebra  $A$ , the intensional algebra  $Flu(A)$  is the intensional algebra whose universe is the cross product of the universe of  $Lul(A)$

together with that of  $\mathbf{Flo}(A)$ ; and which preserves the meaning of the two algebras ( $\mathbf{Flo}(A)$  and  $\mathbf{Lu}(A)$ ) over such a universe. For example,

let the universe of  $\mathbf{Flo}(A)$  be  $P$ , and the universe of  $\mathbf{Lu}(A)$  be  $\omega$ ,  
then the universe of  $\mathbf{Flu}(A)$  is  $\omega \times P$ .

The Lucid operator symbol **first** is defined in  $\mathbf{Lu}(A)$  by the equation:

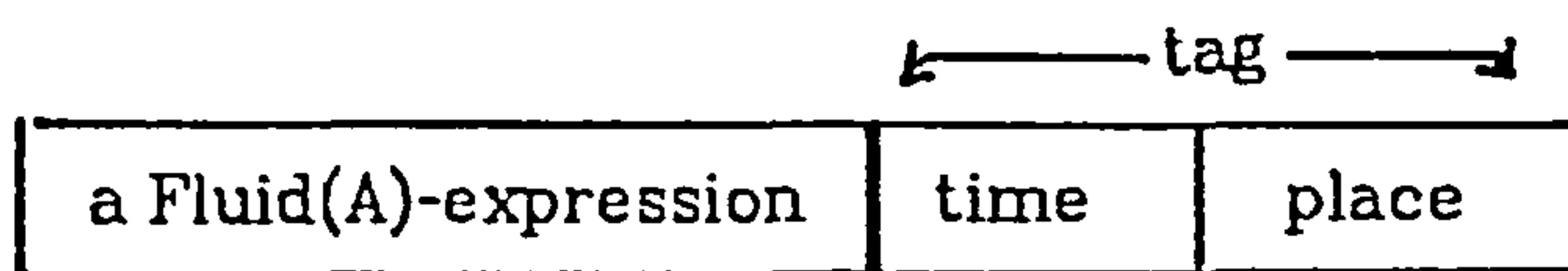
$$(\mathbf{Lu}(A)(\mathbf{first}(x)))_n = (\mathbf{Lu}(A)(x))_0 \text{ for any } n \in \omega \text{ and any } \mathbf{Lu}(A)\text{-expression } x.$$

The meaning of **first** in  $\mathbf{Flu}(A)$  is defined by the equation:

$$(\mathbf{Flu}(A)(\mathbf{first}(x)))_{\langle n, p \rangle} = (\mathbf{Flu}(A)(x))_{\langle 0, p \rangle} \text{ for any } n \in \omega, \text{ any } p \in P,$$

and any expression  $x$  in  $\mathbf{Flu}(A)$

In Chapter 6 also, we describe the techniques for evaluating Fluid. We give a set of rewrite rules for the algebra  $\mathbf{Flu}$  and describe, by means of an example, the reduction approach for evaluating the family of target languages **Fluid**. Moreover, we describe the education approach to evaluating Fluid. In such an approach, the tag should consist of two cells; the time cell which may be manipulated by the intensional operators of  $\mathbf{Lu}$  and the place cell which may be manipulated by those of  $\mathbf{Flu}$ . The reader should note that the tag as a whole is an element (a world) in the universe of the algebra  $\mathbf{Flu}$ ; i.e. it is a pair consisting of an instant in time and a list with back pointer. Thus a tagged expression in the evaluation of  $\mathbf{Fluid}(A)$  will look like



Finally, in Chapter 7, we shed some light on possible extensions for some of the concepts which this thesis has discussed, and suggest lines for further research on the subjects related to this thesis.

### 0.6.2- Mathematical Notation:

In this thesis we use the von Neumann set theoretic representation of natural numbers. In such a representation,

the number 0 is the empty set

the number 1 is the set  $\{0\}$

the number 2 is the set  $\{0,1\}$

the number  $n$  is the set  $\{0,1,2,\dots,n-1\}$

and so on. This process can be extended to represent all the infinite ordinals.

Our interest in this thesis is only with the set of natural numbers which is the first infinite ordinal  $\omega$ . Therefore, by the set of natural numbers  $\omega$  we mean the set  $\{0,1,2,3,\dots\}$ .

Given the sets  $A$  and  $B$ , a relation  $R$  between  $A$  and  $B$  is a subset of the cartesian product of  $A$  and  $B$ .

That is,  $R \subset A \times B$

A function  $F$  from  $A$  to  $B$  is (by definition) a relation in which, for every  $a$  in  $A$  there exist only one element  $b$  in  $B$  such that the pair  $\langle a,b \rangle$  is in  $F$ .

That is,  $\forall a \in A \exists ! b \in B$  such that  $\langle a,b \rangle \in F$

Thus a function throughout this thesis is a set of ordered pairs.

We use the conventional set building notation  $\{ \dots \mid \dots \}$ . For example, the domain of the function  $F$  above, denoted by  $\text{dom}(F)$ , and the range  $\text{range}(F)$  can be defined respectively as

$$\text{dom}(F) = \{ a \mid \langle a,b \rangle \in F \}$$

$$\text{range}(F) = \{ b \mid \langle a,b \rangle \in F \}$$

Given the sets  $A$  and  $B$ , the set  $[A \rightarrow B]$  is the set of all functions from  $A$  to  $B$ . Sometimes we shall write  ${}^A B$  to denote such a set. We use these two representations to make our notation simpler and more readable. For example, instead of writing  $({}^A B)_C$  or  $[[A \rightarrow B] \rightarrow C]$  to denote the set of functions from



$A^B$  to  $C$ , we write  $[(^A B) \rightarrow C]$ .

Given a set  $A$ , the set of finite and infinite sequences over  $A$  is defined to be the set of all functions from  $\omega$  to  $A$ . Thus it is the set represented by  ${}^A A$ .

The reader should note that a function from a natural number  $n$  to a set  $X$  is that function which maps the set  $\{0,1,2,\dots,n-1\}$  to  $X$ .

We use the angular brackets as a sequence builder notation, for example the sequence  $\langle 2*i : i \in \omega \rangle$  is the sequence of even integers.

Or we use the explicit form and write

$$\langle 2,4,6,8, \dots \rangle.$$

We use two ways to represent a certain element in a sequence. Either the function application notation; for example, the first element in the sequence  $X$  is written as  $X(0)$ . Or we use the indexing notation and write  $X_0$ . Sometimes, we mix these two representations to make our notation more readable. For example, if  $X$  is a hyper-sequence (a sequence of sequences) then the  $i$ 'th element of the  $j$ 'th sequence can be represented by  $(X_j)_i$  or by  $(X(j))(i)$ . However, for clarity we shall write  $X(j)_i$ .

While a sequence is a function whose domain is a natural number, a "family" is a function whose domain is not necessarily well ordered. We use such a concept (the family) to define families of languages (such as ISWIM and LUCID) and algebras (such as  $\mathcal{L}\mathcal{U}$ ). For example, we say that ISWIM is a family of languages in which each member is determined by an algebra of data types. We write

$$\text{ISWIM} = \{\text{ISWIM}(A) : A \text{ is an algebra of data types}\}.$$

Similarly,  $\mathcal{L}\mathcal{U}$  is the family of algebras

$$\{\mathcal{L}\mathcal{U}(A) : A \text{ is an algebra of data types}\}.$$

We use function application to represent a certain member of a family. Thus,  $\text{ISWIM}(Z)$  is the member of ISWIM whose algebra of data types is  $Z$ .

# Chapter 1

## The Language Lucid and Its Implementations

### 1.0- Languages and Algebras:

Lucid is a family of functional programming languages [AsWa79, AsWa80, WaAs84]. Each member of such a family is determined by a choice of underlying data structures. That is, each member is determined by a domain of data objects together with a collection of primitive operations on these objects. Thus, when talking about a certain member of the Lucid family, we have to specify the collection of data objects which such a member embodies. For example, if  $N$  is the algebra of natural numbers ( $N$  is the set of natural numbers together with the usual mathematical operations) represented by their usual symbols, then  $\text{Lucid}(N)$  is a member of Lucid. Hence, when running a program in  $\text{Lucid}(N)$ , the output of the machine will be elements in such a data type - i.e. natural numbers.

The idea of a language being a function of an algebra (or a family of programming languages) was first introduced by P.Landin in the definition of ISWIM [Lan66]. To quote Landin in [Lan66] "Iswim is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives". So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives". What Landin meant by "a set of primitives" is exactly what had been formalized afterwards as an algebra of data types.

Informally speaking, a data type is a domain of data objects together with a collection of operations defined on these objects. For example, the set of

natural numbers together with the usual mathematical operations on them; or the set of truth values with the usual boolean operations, and so on. Such mathematical structures are often called *abstract data types* to emphasize the fact that we are talking about the abstract mathematical structure and properties without having any particular representation in mind. An algebra is, simply, a representation of abstract data types.

To construct an algebra, or a representation for an abstract data type, we need a collection of constant symbols of various arities. This is called the **signature** of the algebra. The nullary symbols in the signature are called individual constants and will represent the data elements of the type. Those of non zero arities are called operator symbols and will represent the operations defined on the elements of the data type. Thus a signature is a set of constant symbols with different arities.

Given a signature  $\Sigma$ , a  $\Sigma$ -algebra  $A$  (or an algebra over  $\Sigma$ ) is a pair  $\langle D, F \rangle$ , where  $D$  is a domain of objects and  $F$  is a function assigning to every  $n$ -ary operator symbol in  $\Sigma$  a function of degree  $n$  on the domain  $D$ . By a domain, we mean a CPO (a chain-complete partially ordered set with the least object  $\perp$ ). Thus, the function  $F$  assigns to the 0-ary constant symbols, of the signature, elements in the set  $D$ . To a non-nullary constant symbol of  $\Sigma$ ,  $F$  assigns an operation on  $D$ ; the arity of such an operation equals that of the symbol.

### Examples:

Let  $\Sigma$  be the set of nullary symbols **tt**, **ff**, and  $\perp$  together with the binary symbols  $\vee$ ,  $\wedge$  and the unary symbol **not**. That is,

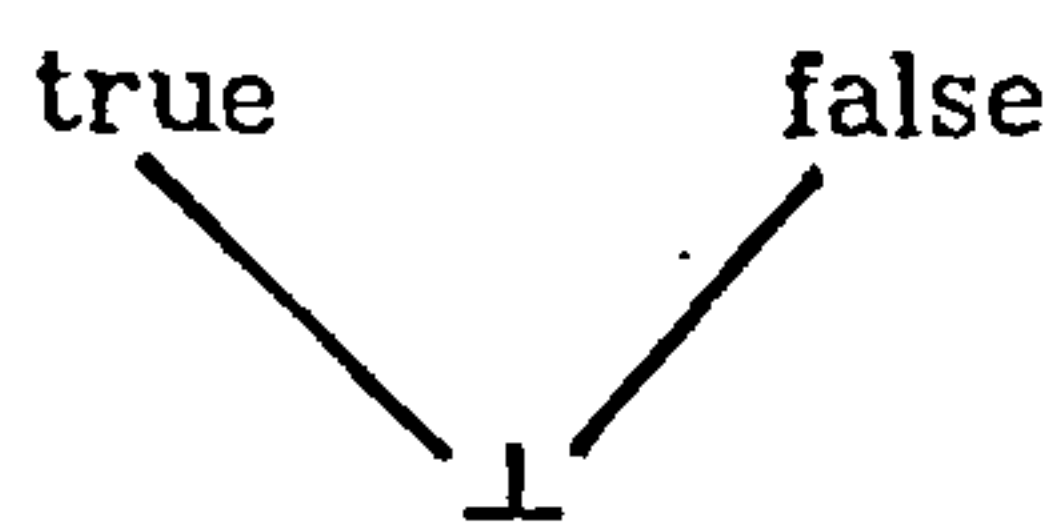
$$\Sigma = \{\text{tt}, \text{ff}, \perp, \vee, \wedge, \text{not}\}$$

Define the algebra  $B = \langle D, F \rangle$  as follows,

let  $D$  be the domain of truth values with the special object  $\perp$ ,



that is,  $D$  is the domain



define  $F: \Sigma \dashrightarrow [D^n \dashrightarrow D]$

such that  $tt \dashrightarrow \text{true}$

$\text{ff} \dashrightarrow \text{false}$

$\perp \dashrightarrow \perp$

And let  $F$  assign the negation function on truth values to the unary symbol *not*, the binary boolean function *and* to the symbol  $\wedge$ , and the binary function *or* to the symbol  $\vee$ . Clearly, the  $\Sigma$ -algebra  $A$  is a representation of the boolean data type.

Another simple example, which we shall use very often in our thesis, is that of the natural numbers.

Let the signature  $\Sigma'$  be the set of nullary constant symbols (the digits)

$\{0, 1, 2, 3, 4, \dots\}$

together with the binary symbols  $-$ ,  $+$ ,  $*$  and *div*. That is,

$\Sigma' = \{-, +, *, \text{div}, 0, 1, 2, 3, 4, \dots\}$

Define the algebra  $A$  over  $\Sigma'$  to be the function which assigns to the symbols of the signature their usual mathematical interpretation in the set of natural numbers. That is, consider the domain of the algebra to be the set of natural numbers, assign the numbers  $0, 1, 2, 3, \dots$  to the digits  $0, 1, 2, 3, \dots$ ; and assign the usual mathematical operations  $+$ ,  $-$ ,  $*$ ,  $\text{div}$  to the symbols  $+$ ,  $-$ ,  $*$ ,  $\text{div}$ . The algebra defined, then, is a representation of the natural numbers. Clearly, it is the algebra of the natural numbers represented by their usual mathematical symbols.

Algebras are not always homogeneous; that is, representing one particular data type only like the natural numbers, the booleans or the rationals. An algebra can represent more than one data type, like the algebra of the natural numbers together with the booleans, the lists with the integers, or the rationals with lists, and so on. There are different ways and approaches to formalize these algebras. Either we use the **many sorted algebra** approach of the ADJ group [GTW78], or we use the **classified algebra** approach of W.Wadge [Wad82]. In the former, we consider the algebra as being sorted to subalgebras. In other words, the domain of the algebra consists of a family of separate subsets, and on each of these subsets there is a collection of operators. There are, also, a set of operators on the set of sorts as a whole. In the classified algebra approach, the domain consists of one single sort; nevertheless, the domain of such a sort is classified (not partitioned like the former) into subclasses.

In this thesis, we adopt the latter approach (the classified algebra approach of W.Wadge). Moreover, we consider that any algebra contains at least the algebra of truth values. Thus a domain is a CPO which at least contains the domain of truth values.

Landin's idea, of a language being a family, was later formalized and carried out by E.Ashcroft and W.Wadge in the design of Lucid. In this thesis, we adopt such a philosophy. Therefore we talk about ISWIM and Lucid as families of programming languages. Each member in these families is a programming language in its own right and determined by the set of primitives or, formally, by the algebra of data types it embodies. Thus, to represent such families we write

$$\text{ISWIM} = \{ \text{ISWIM}(A) : A \text{ is an algebra of data types} \}$$

$$\text{Lucid} = \{ \text{Lucid}(A) : A \text{ is an algebra of data types} \}$$

The reader is reminded that, the different members of a family are identical in their outer appearance. For example, all members in the Lucid

family use the **where** clause for nesting texts and scopes; they also use the Lucid operators (such as **first**, **next**, ...etc). Therefore, when defining a family of languages, our interest is mainly with defining the collection of parse trees and not the collection of strings. The collection of strings is dependent on the set of primitives (or the data algebra); hence, it is a property of each particular member. Thus, to define a family of languages, we have to give the abstract syntax and not the concrete one.

For example, an attempt to give a BNF-like grammar to the family Lucid will yield

```

<program> ::= <expression>

<expression> ::= <simple expression>
                | <where expression>

<where expression> ::= <expression> where
                        <declarations> <definitions list>
                        end

```

The meta-variable <expression>, which forms the heart of the language cannot be defined without knowing the algebra of data types we are considering. We want to be able to say that "an n-ary operator symbol in the algebra applied to n expressions in the language is also an expression".

Notice, however, that while the abstract syntax is used to define a family, any member of the family can be described using a BNF-like formalism with some annotations. We shall use the abstract syntax to define the languages discussed in this thesis.



### 1.1- Lucid, the Dataflow Functional Language:

*Iteration* is an important operational concept in programming. It, simply, captures the dynamicity of the computation process. Due to the explicit control structure of the von Neumann languages and the "variable-storage cell" relation, this concept was easy to mimic via the assignment statement.

Functional languages however, are descriptive rather than imperative. A program is an expression specifying the result rather than a sequence of assignments to be executed. As a result of losing the explicit control in functional languages we lost also the ability to express iterative algorithms. Iteration was replaced by a cleaner and a more mathematically acceptable concept *recursion*.

Even though 'recursion' enabled us to represent some loop-based iterative algorithms, it could not fully replace 'iteration'. It would be both unnatural and difficult to use 'recursion' for expressing some simple iterative algorithms. Hence, if we want functional languages to have a more expressive power, we should incorporate 'iteration' as well as 'recursion' in these languages.

Apart from Lucid, functional languages (such as KRC and Lispkit) can imitate iteration by incorporating infinite lists as a data structure. In Lucid, however, iteration is explicit via the algebra of histories *Lu*. It is this property of Lucid (explicitly iterative language) which makes it a proper language for dataflow machines.

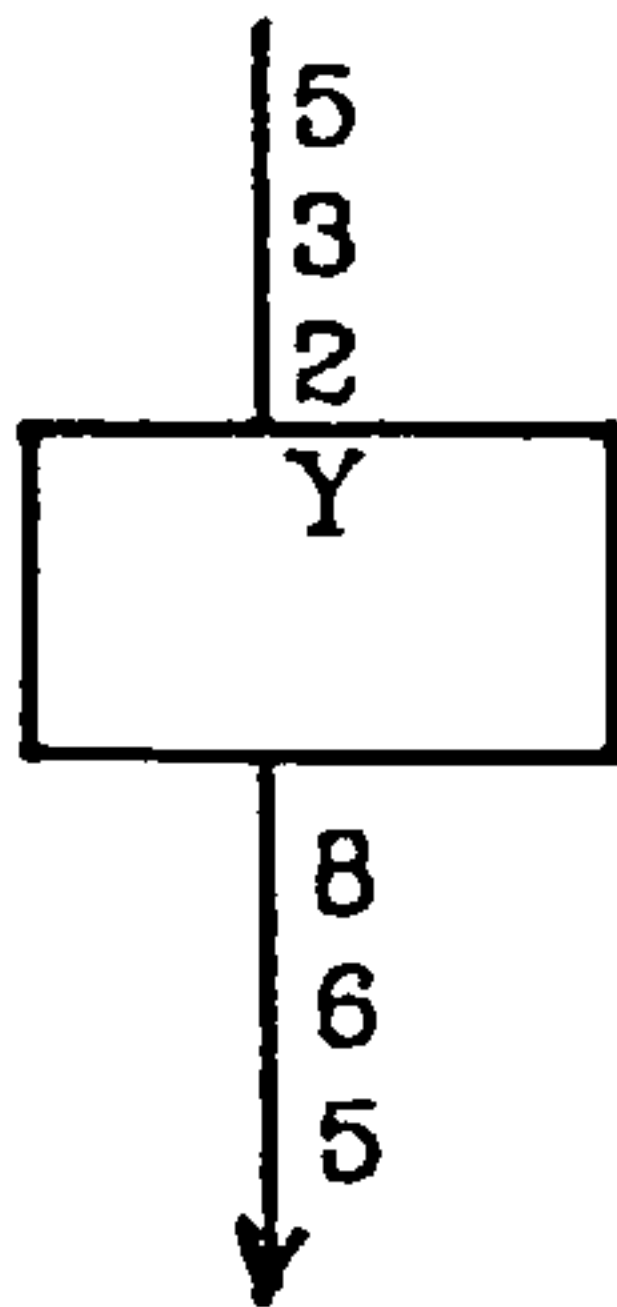
Moreover, Lucid is unlike other functional languages in the sense that a program in Lucid does not produce one single data object, but is a continuously operating machine. For example, the machine running the Lucid(N) expression

**X where X = 3; end**

acts like a box which keeps producing the value 3 at its output port. The machine running the Lucid(N) expression

`X + Y where X = 3; end`

acts like a filter with two ports; one for the input and one for the output. It takes an input value and outputs the sum of such a value plus 3.



Therefore, we talk about the value of an expression as being a history which is the infinite sequence of values which the expression takes during evaluation.

Denotationally, we represent such histories by infinite sequences. That is, we talk about the overall value of a Lucid expression as being an infinite sequence which represents the subsequent values the expression takes during execution. Therefore, a program produces an infinite sequence of data objects, and its inputs (if there are any) are also infinite sequences of objects. For example, the value of the expression

`X where X=3; end`

is the constant infinite sequence  $\langle 3, 3, 3, 3, \dots \rangle$ .

Also, given the sequence  $\langle 2, 3, 5, 4, \dots \rangle$  as a value for the variable symbol `Y` in the expression

`X+Y where X=3; end`

the value of the expression is the infinite sequence  $\langle 5, 6, 8, 7, \dots \rangle$ .

Even though the meanings (denotations) of expressions in Lucid are infinite sequences, Lucid is not intended to be a language for manipulating sequences. That is, it is not a usual functional language (such as KRC and Lispkit for example) with the set of sequences as a domain. When programming in Lucid, it is advisable to think in terms of dynamically changing objects (or quantities

which change with time) rather than infinite sequences.

For example, suppose we want to write a program to generate the natural numbers. In PASCAL (for example), we start by assigning the number 0 to a variable symbol, say *X*. Then we use the **while-loop** construct to repeatedly increment the value of *X* by 1. Thus, we write

```

program infinite(input,output);
var X:integer;
begin
  X := 0;
  while X >= 0 do begin
    writeln(X);
    X := X+1;
  end;
end.

```

The reader should notice that the dynamicity of incrementing *X* is given by the **while** construct together with the assignment statement *X*:=*X*+1.

Lucid, however, facilitates dynamicity via a collection of temporal operators- the Lucid operators. For example, one of these operators is **fb** (followed by). The definition

$$Z = 0 \text{ fby } 1$$

means, informally, that the value of *Z* is initially set to 0 then it takes the value 1 afterwards. Thus the initial value of the expression

$$X \text{ fby } Y \text{ where } X = 3 \text{ and } Y = 5$$

is 3, then it is 5 afterwards.

Using **fb**, we can recursively generate the infinite stream of natural numbers. We set the initial value of the expression to 0 then recursively increment such a value by 1. That is, we write

```

X where
  X = 0 fby X+1;
end

```



Later in this chapter we shall give a brief description of the other Lucid operators.

Another useful and natural way of thinking about Lucid programs is in terms of dataflow nets - or *plumbing*. Both the user-defined functions and the operators of the data algebra (such as  $+$ ,  $-$ ,  $\text{div}$  in the algebra  $N$ ) are thought of as nodes or continuously operating filters. The variable symbols represent the arcs of the net or the communication channels. For example, the following Lucid( $N$ ) program generates the infinite sequence of squares

$\langle 0, 1, 4, 9, 16, \dots \rangle$

**square(n) where**

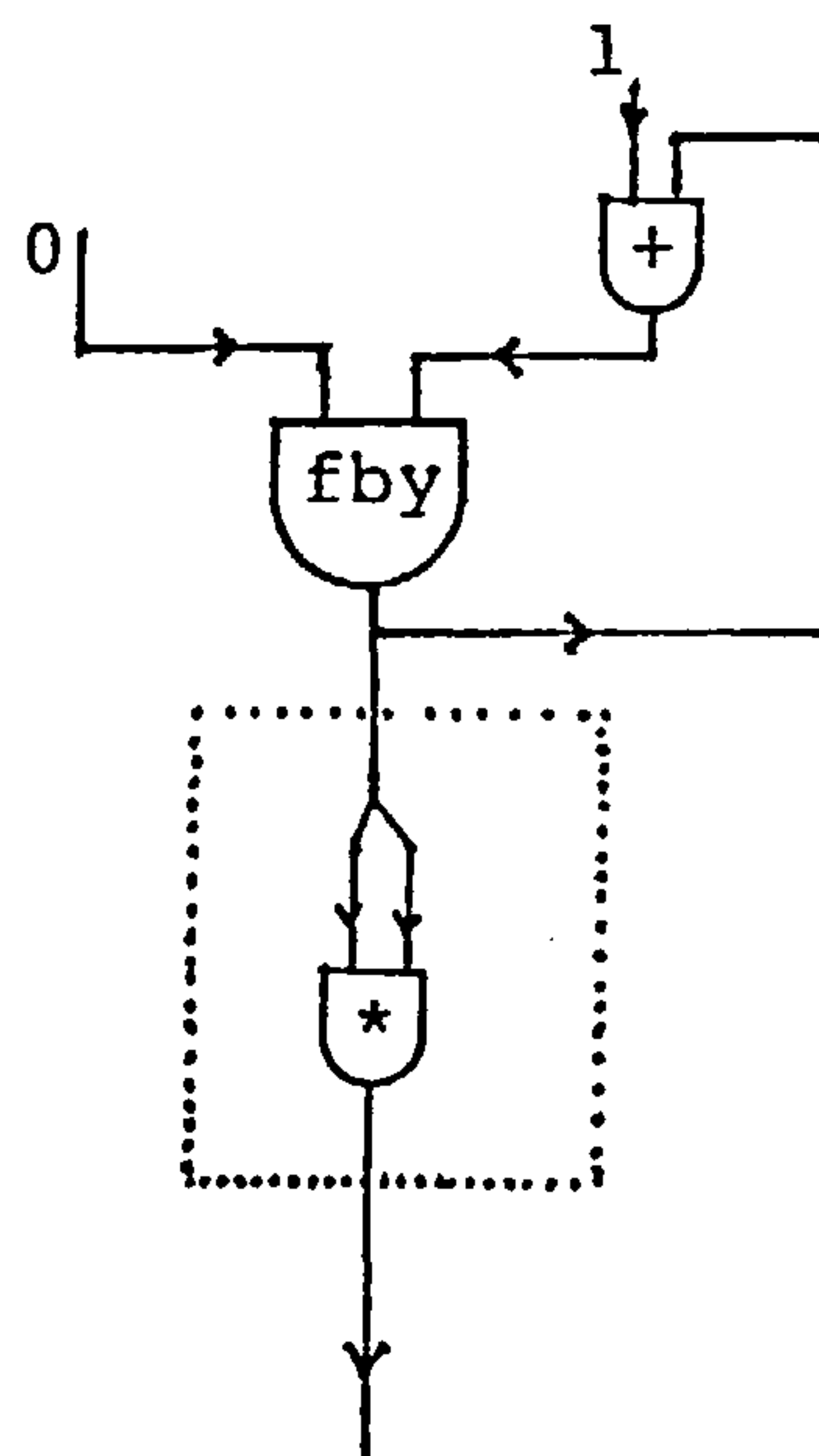
**square(X) = X \* X;**

**n = 0 fby n + 1;**

**end**

The program can be represented by the net

square



The reader should notice that the nodes representing the operators **fby**, **+** and the user-defined function **square** act like continuously operating filters. Each take an infinite stream of input tokens and returns an infinite stream of output ones. Moreover, the node representing the user-defined function (**square**) is itself an encapsulated net. That is, it is both a node and a subnet in the main graph.

## 1.2- The Family of Languages Luswim:

From the above discussion, the reader should notice two main aspects of Lucid:

First, it is a structured definitional functional language; a program is an expression. An expression is either simple, such as

$$X+Y, (A*X*X+B*X-C) / A*B ;$$

or a *where* clause with an expression as a *subject* (the left hand side of *where*) and a set of auxiliary definitions as a *body* (enclosed between *where* and *end*) such as

$(A * \text{square}(X) + B * X - C) / A * B$  *where*

$\text{square}(X) = X * X;$

$A = 3;$

$B = 5;$

$X = 10 * Y;$

*end*

The second main aspect of Lucid is that, it is a language over histories (denoted by infinite sequences) of data objects.

The first property of Lucid is due to the language Iswim which is a variant of Landin's ISWIM, while the second is facilitated by the algebraic function  $Lu$ . Given a data algebra  $A$ , the algebra  $Lu(A)$  is the algebra of histories of objects of  $A$ . A description of  $Lu(A)$  will be given later.

Formally then, we can say that Lucid is Iswim with iteration. That is, it is the subfamily of Iswim whose algebra of data types is the algebra  $Lu$  of histories. Thus, it is the family  $\text{Iswim}(Lu)$ . However, apart from simple iteration, Lucid has another main aspect which is *currenting-* or *nested iteration*. Thus Lucid is the language  $\text{Iswim}(Lu)$  together with nested iteration. The family of languages  $\text{Iswim}(Lu)$  is called Luswim; and thus

$$\text{Luswim} = \{\text{Iswim}(\text{Lu}(A)) : A \text{ is a data algebra}\}$$

Clearly, Luswim is a subfamily of Lucid; that is, for any algebra  $A$ ,  $\text{Luswim}(A)$  is a member of the family Lucid. In the next two sections we give a formal description of both Iswim and  $\text{Lu}$ .

### 1.2.1- The Language Iswim:

$\text{Uswim}^\dagger$  is a family of languages designed by Ashcroft and Wadge as the basis for Structured Lucid [AsWa80, AsWa79]. It is a variant of Landin's ISWIM<sup>††</sup>, and was designed to provide a general way of allowing structured definitions in Lucid by using the Iswim where clause.

In their latest publication [WaAs84], the inventors of Lucid have changed the name Uswim to Iswim, and to distinguish between Landin's ISWIM and their version of the language they refer to the former as ISWIM. This terminology is used throughout this thesis.

The main differences between Iswim and Landin's ISWIM are two:

First, the where-clause in Iswim corresponds to **whererec** in Landin's. That is, while the latter distinguishes between the non-recursive **where** and the recursive **whererec**, the former assumes **where** to be recursive.

The second difference is that all functions in Iswim are of first order; i.e. no higher order functions are allowed in Iswim. This, in fact, makes Iswim a proper subset of Landin's ISWIM.

In the next section, we give the abstract syntax of the family Iswim. Throughout this thesis, we assume that we have an unlimited number of variable symbols with different arities. These are known in computer science terminology as identifiers. The nullary variable symbols are called individual variables, and

---

† U See What I Mean.

†† If U See What I Mean.



those of arity greater than 0 are called function symbols.

### 1.2.1.A- The Abstract Syntax of Iswim:

Given a  $\Sigma$ -algebra  $A$ , a program in  $\text{Iswim}(A)$  is an  $\text{Iswim}(A)$ -expression. The set of expressions of  $\text{Iswim}(A)$  is the smallest set  $T$  such that:

- (\*) if  $\psi$  is an  $n$ -ary constant symbol in  $\Sigma$  and  $x_0, \dots, x_{n-1}$  are  $n$  expressions in  $T$ , then  $\psi(x_0, \dots, x_{n-1})$  is in  $T$ .
- (\*) if  $\vartheta$  is an  $n$ -ary variable symbol, and  $x_0, \dots, x_{n-1}$  are in  $T$ , then  $\vartheta(x_0, \dots, x_{n-1})$  is in  $T$ .
- (\*) if  $\varepsilon$  is in  $T$ , and  $D$  is a compatible set of definitions in  $\text{Iswim}(A)$ , then  $\varepsilon \text{ where } D \text{ end}$  is in  $T$ .

$\text{Iswim}(A)$ -definitions:

- (\*) A definition in  $\text{Iswim}(A)$  is of the form

$$f(a_0, \dots, a_{n-1}) = \varepsilon$$

where  $f$  is an  $n$ -ary variable symbol,  $a_0, \dots, a_{n-1}$  is a list of distinct nullary variable symbols, called the formal parameters of  $f$ , and  $\varepsilon$  is an expression in  $\text{Iswim}(A)$ .

A set of  $\text{Iswim}(A)$ -definitions is called **compatible** if no two definitions in the set have the same left hand side.

A variable symbol is called **local** to a where-clause expression when it is defined in the body of the clause. Otherwise, it is called **global**.

An occurrence of a variable symbol in an  $\text{Iswim}$ -expression is called **bound** when it refers to a variable which is local in an enclosing where-expression or a

formal parameter of an enclosing definition.

For example, in the program

$F(X+Y,Z)$  where

$F(A,B) = G(A) + V + N$  where

$G(W) = N * W + B;$

$N = 3;$

end;

$V = X + N;$

$X = 10 + Z;$

end

the variable symbols  $F$ ,  $V$ , and  $X$  are all local to the main (outer) where-clause. Thus the occurrences of these variable symbols, and the occurrences of the formals  $A$ , and  $B$  are bound occurrences in the clause.

The occurrence of  $N$  in the right hand side of the definition of  $F$  in the inner clause is bound as  $N$  is local to the clause. However, the occurrence of  $N$  in the outer clause is free because it is not defined in the clause. In other words, the scope of definitions propagates inwards. Also, the occurrence of  $W$  in the definition of  $G$  is also bound as it is the formal parameter of the definition.

#### 1.2.1.B- The Semantics of Iswim:

Given an algebra  $A$ , the semantics of  $\text{Iswim}(A)$  is uniquely determined by the algebra  $A$ . Like any other programming language however, the meaning of an expression in  $\text{Iswim}(A)$  depends also on the environment of the evaluation. The reader should note that, while the algebra assigns meaning to the constant symbols of the language, the environment assigns meaning to the set of variable symbols.

An environment is a function which assigns to every  $n$ -ary variable symbol a

function of degree  $n$  on the domain of the algebra  $A$ .

Given an algebra  $A$ , the value of an  $\text{Iswim}(A)$ -expression  $\varepsilon$  in an environment  $\varphi$ , written here as  $\models_{A,\varphi} \varepsilon$ , is defined recursively as follows:

(\*) if  $\varepsilon$  is of the form  $\psi(x_0, \dots, x_{n-1})$  where  $\psi$  is an  $n$ -ary constant symbol and  $x_0, \dots, x_{n-1}$  are  $n$   $\text{Iswim}(A)$ -expressions, then

$$\models_{A,\varphi} \psi(x_0, \dots, x_{n-1}) = A(\psi)(\models_{A,\varphi} x_0, \dots, \models_{A,\varphi} x_{n-1})$$

(\*) if  $\varepsilon$  is of the form  $\vartheta(x_0, \dots, x_{n-1})$  where  $\vartheta$  is an  $n$ -ary variable symbol and  $x_0, \dots, x_{n-1}$  are  $n$   $\text{Iswim}(A)$ -expressions, then

$$\models_{A,\varphi} \vartheta(x_0, \dots, x_{n-1}) = \varphi(\vartheta)(\models_{A,\varphi} x_0, \dots, \models_{A,\varphi} x_{n-1})$$

(\*) if  $\varepsilon$  is of the form  $E$  where  $D$  end

where  $E$  is an  $\text{Iswim}(A)$ -expression and  $D$  is a compatible set of  $\text{Iswim}(A)$ -definitions, then

$$\models_{A,\varphi} \varepsilon = \models_{A,\varphi'} E$$

where  $\varphi'$  is the least environment which satisfies the set of definitions  $D$  and agrees with  $\varphi$  except at most on the values assigned to the locals of  $\varepsilon$ .

The reader should notice here that because the domain of the algebra  $A$  is a CPO, it induces a partial order on the set of all  $\text{Iswim}(A)$ -environments. That is, if  $A$  is the domain of the algebra  $A$ , then for any  $\text{Iswim}(A)$ -environments  $\varepsilon, \varepsilon'$

$$\varepsilon < \varepsilon' \Leftrightarrow \forall v \in V (\varepsilon(v) <_A \varepsilon'(v)) \quad \text{where } V \text{ is the set of variable symbols.}$$

#### Definition:

An environment  $e$  satisfies a definition  $d$  if and only if, for any environment  $e'$  which agrees with  $e$  except possibly on the values assigned to the formal parameters of the definition, the value of the right hand side of the definition equals the value of its left hand side.

An environment satisfies a set of definitions if it satisfies every definition in the set simultaneously.



## 1.2.1.C- Examples:

The following is an expression, hence a program, in  $\text{Iswim}(\text{list})$  where  $\text{list}$  is the algebra of lists represented by their usual symbols

$\text{invert}(\text{input})$  where

$\text{invert}(L) = \text{if } L \text{ eq nil then nil}$

$\text{else invert}(\text{tl}(L)) \text{ <> } (\text{hd}(L) :: []) \text{ fi};$

$\text{end}$

The meaning of the program above in an environment  $\mathcal{V}$  is a unary function. Given the value  $\mathcal{V}(\text{input})$  (the value which the environment  $\mathcal{V}$  assigns to the variable symbol  $\text{input}$ ), the function returns the inverse of such a value if it is a list. Otherwise, it returns the bottom element of the domain (i.e.  $\perp$ ).

Another program in  $\text{Iswim}(\mathbb{N})$  is the following which computes the 10th prime number:

$\text{prime}(10)$  where

$\text{prime}(N) = \text{if } N \text{ eq } 1 \text{ then } 2$

$\text{else nextprime}(\text{prime}(N-1)) \text{ fi};$

$\text{nextprime}(M) = \text{if isprime}(M+1) \text{ then } M+1$

$\text{else nextprime}(M+1) \text{ fi};$

$\text{isprime}(P) = \text{notdivgt}(2) \text{ where}$

$\text{notdivgt}(Q) = \text{if } Q * Q > P \text{ then true}$

$\text{elseif } (P \bmod Q \text{ eq } 0) \text{ then false}$

$\text{else notdivgt}(Q+1) \text{ fi};$

$\text{end};$

$\text{end}$

### 1.2.2- The Family of Algebras $\mathcal{L}\mathcal{U}$ :

$\mathcal{L}\mathcal{U}$  is a function on algebras; for any algebra  $A$ ,  $\mathcal{L}\mathcal{U}(A)$  is uniquely determined by the algebra  $A$ . Thus  $\mathcal{L}\mathcal{U}$  is the family of algebras

$$\{\mathcal{L}\mathcal{U}(A) : A \text{ is an algebra}\}$$

Given a data algebra  $A$  over a signature  $\Sigma$ ,  $\mathcal{L}\mathcal{U}(A)$  is the algebra defined as follows

- the domain of  $\mathcal{L}\mathcal{U}(A)$  is the domain of  $\omega$ -histories (or infinite sequences) of elements of the domain of  $A$ . The partial order on such a domain is the pointwise extension of the partial order of the domain of  $A$ .
- $\mathcal{L}\mathcal{U}(A)$  extends the operations of  $A$  (the meanings which are given to the symbols of  $\Sigma$  by the algebra  $A$ ) pointwise over the domain of histories.

For example, the domain of the algebra  $\mathcal{L}\mathcal{U}(N)$ , where  $N$  is the algebra of natural numbers described before, is the set of infinite sequences of natural numbers. The operator symbols of  $N$  (such as  $+$ ,  $-$ , and  $\text{div}$ ) are interpreted pointwise on such a domain. Thus,

$$\langle 2, 3, 6, 3, \dots \rangle + \langle 4, 6, 0, 5, \dots \rangle = \langle 6, 9, 6, 8, \dots \rangle$$

$$\langle 4, 5, 6, 2, \dots \rangle - \langle 2, 8, 6, 1, \dots \rangle = \langle 2, 1, 0, 1, \dots \rangle$$

and so on.

Apart from extending the operators of  $A$  pointwise over the set of  $\omega$ -histories of the domain of  $A$ ,  $\mathcal{L}\mathcal{U}(A)$  assigns meanings to the set of constant symbols (the Lucid operator symbols)<sup>†</sup>

$\{\text{first}, \text{next}, \text{fby}, \text{whenever}, \text{asa}, \text{upon}, \text{attime}\}$

Except  $\text{first}$  and  $\text{next}$  which are unary operator symbols, each of the symbols above is of arity 2.

Informally speaking, the operations which  $\mathcal{L}\mathcal{U}$  assigns to these symbols are as follows: (for simplicity, we shall write the meaning  $\mathcal{L}\mathcal{U}(X)$  of the operator

We assume here that the signature  $\Sigma$  of  $A$  does not contain any Lucid operator symbol.

symbol  $X$  as  $X$ . For example, the operation  $Lu(\text{first})$  is written as  $\text{first}$ )

$Lu(\text{first})$  : is a unary operation which returns the constant sequence of the first element of its argument. For example

$$\text{first}(\langle 2, 5, 2, 8, \dots \rangle) = \langle 2, 2, 2, 2, \dots \rangle.$$

$Lu(\text{next})$  : is a unary operation which returns its argument sequence without the first element. For example

$$\text{next}(\langle 4, 3, 5, 6, 1, \dots \rangle) = \langle 3, 5, 6, 1, \dots \rangle.$$

$Lu(\text{fby})$  : (followed by) is a binary operation which returns the first element of the first argument sequence concatenated by the second argument sequence.

$$\text{fby}(\langle 2, 4, 6, 8, \dots \rangle, \langle 1, 3, 5, 7, \dots \rangle) = \langle 2, 1, 3, 5, 7, \dots \rangle$$

$Lu(\text{whenever})$  : is a binary operation which returns the component of its first argument sequence whenever the corresponding component of the second sequence is the object `true`.

$$\text{whenever}(\langle 3, 5, 7, 9, \dots \rangle, \langle \text{tt}, \text{ff}, \text{ff}, \text{tt}, \dots \rangle) = \langle 3, 9, \dots \rangle$$

$Lu(\text{asa})$  : (as soon as) is a binary operation which checks the components of its second argument sequence. Once it finds the object `true`, it returns the constant sequence of the corresponding component in its first argument

$$\text{asa}(\langle 3, 5, 7, 9, \dots \rangle, \langle \text{ff}, \text{ff}, \text{ff}, \text{tt}, \dots \rangle) = \langle 9, 9, 9, 9, \dots \rangle$$

$Lu(\text{upon})$  : (advance upon true) is a binary operation. It is easy to understand such an operation as a node with two input ports and one output. The first component of its output is the first component of its first argument sequence; i.e. it lets the first component of its first argument pass as an output. Then it checks the components of its second argument one by one. If a component is the object `false` then it repeats its previous output; if the component is `true` it



lets another element from the first argument pass as an output. In other words, this operation acts like a filter which advances (one-at-a time) the components of its first input only when it finds the object true in the second argument. For example

$$\text{upon}(\langle 0, 2, 4, 6, 8, 10, \dots \rangle, \langle \text{tt}, \text{ff}, \text{tt}, \text{ff}, \text{tt}, \text{tt}, \dots \rangle) = \langle 0, 2, 2, 4, 4, 6, \dots \rangle$$

$\text{Lu}(\text{at time})$  : (at time) is a binary operation which returns the n'th component of the first argument sequence, where n is the first element of the second argument <sup>†</sup>.

$$\text{at time}(\langle 3, 6, 8, 12, 23, \dots \rangle, \langle 3, 5, 6, 2, \dots \rangle) = \langle 12, 12, 12, \dots \rangle$$

We give here the formal definition of the algebraic function (the family of algebras)  $\text{Lu}$ :

**Definition:**

Let  $A (= \langle F, D \rangle)$  be an extensional  $\Sigma$ -algebra. Then  $\text{Lu}(A)$  is the algebra over the signature

$$\Sigma \cup \{\text{first}, \text{next}, \text{fby}, \text{whenever}, \text{asa}, \text{upon}, \text{at time}\}$$

and whose domain is  ${}^{\omega}D$ , with the pointwise extension of the partial order defined on  $D$ , such that:

(a):  $\text{Lu}(A)$  extends the operations of  $A$  pointwise:

That is, for every n-ary constant symbol  $\psi$  in  $\Sigma$ ,

for every n  $\text{Lu}(A)$ -terms  $x_0, \dots, x_{n-1}$

$$\text{Lu}(A)(\psi(x_0, \dots, x_{n-1})) = \lambda i \in \omega [F(\psi)(\text{Lu}(x_0)(i), \dots, \text{Lu}(x_{n-1})(i))]$$

(b):  $\text{Lu}(A)$  assigns meaning to the Lucid operation symbols as

follows

For every  $x, y, t$  in  ${}^{\omega}D$

---

The reader is reminded that the first natural number, in our convention, is 0.

$$L\mu(A)(first(x)) = \lambda i \in \omega [x_0]$$

$$L\mu(A)(next(x)) = \lambda i \in \omega [x_{i+1}]$$

$$L\mu(A)(fby(x,y)) = \lambda i \in \omega [if (i \text{ eq } 0) \text{ then } x_i \text{ else } y_{i-1}]$$

$$L\mu(A)(asa(x,t)) = \lambda i \in \omega [if \exists j \leq i \ t_j \text{ and } (k \leq j \rightarrow \text{not } t_k) \\ \text{then } x_j \text{ else } \perp]$$

$$L\mu(A)(upon(x,t)) = \lambda i \in \omega [x_n] \quad \text{where } n = \text{Card } \{j \leq i : t_j\}$$

and Card is the cardinality function on sets.

$$L\mu(A)(attime(x,t)) = \lambda i \in \omega [x_{t_i}] \text{ or equivalently } \lambda i \in \omega [x_{t(i)}]$$

$$L\mu(A)(whenever(x,t)) = \lambda i \in \omega [if \ t_i \text{ then } x_i \text{ fby } (whenever(next(x),next(y)),_i \\ \text{else } (whenever(next(x),next(y)),_i)]$$

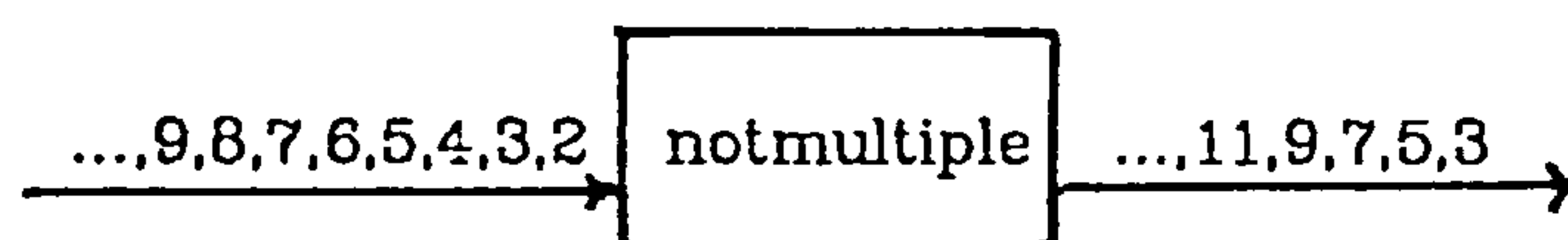
where first =  $L\mu(A)first$

where next =  $L\mu(A)next$

where whenever =  $L\mu(A)whenever$

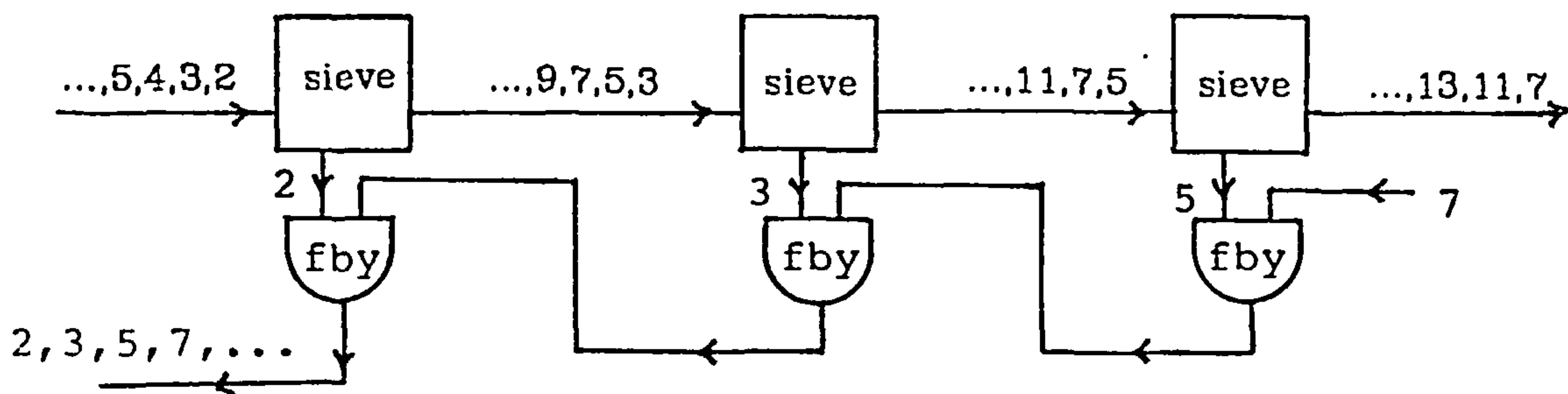
### 1.2.3- Examples:

1- The following is a Luswim(N) program which generates the infinite sequence of prime numbers using the sieve of Eratosthenes. The main parts of the algorithm are the functions notmultiple and sieve. The function notmultiple, when applied to a sequence of natural numbers, returns the sequence after filtering out all the multiples of the first element.



The recursive function sieve returns the first element of its argument sequence followed by the sequence resulting from sieving (filtering) the multiples of such a number from the sequence. The algorithm can be illustrated as an infinite sequence of boxes (each representing a copy of the function sieve). Each box takes a stream of natural numbers, produces the first element of the stream as

an output then passes the stream to the next box.



The program can be written in Luswim(N) as:

**sieve(N) where**

**N = 2 fby N+1;**

**sieve(X) = X fby sieve(notmultiple) where**

**notmultiple = X wvr (X mod first X ne 0);**

**end;**

**end**

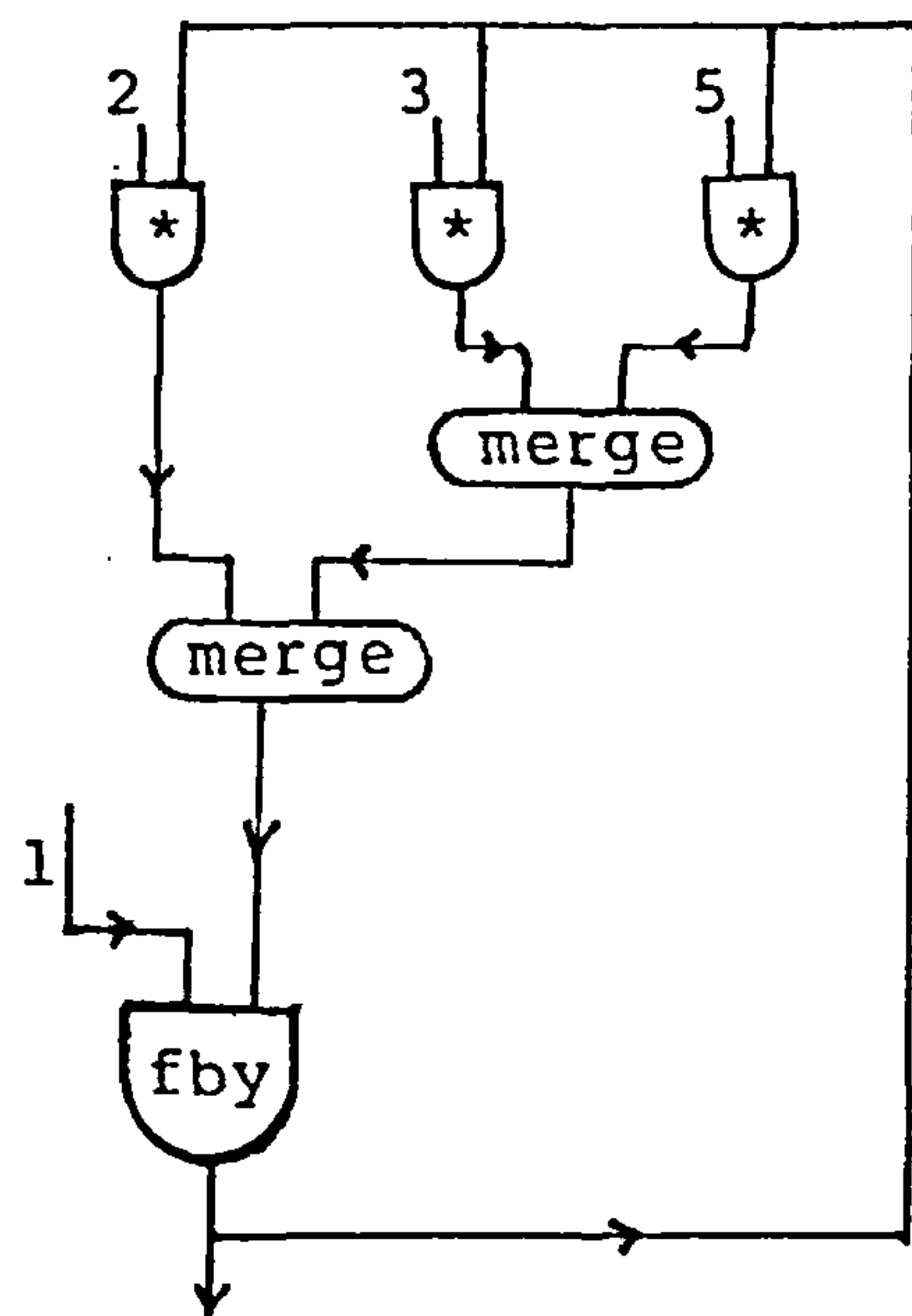
## 2- The Hamming Problem:

The problem is to generate, in an increasing order, all the natural numbers whose prime factors are 2, 3, and 5. The sequence of such numbers is defined recursively as follows:

1 is an element in the sequence.

if  $n$  is in the sequence, then so are  $2*n$ ,  $3*n$ , and  $5*n$

Clearly, the first number in the sequence is 1. We can generate the rest of the sequence recursively by passing the output (so far) to three nodes. One multiplies its input by 2, the other by 3 and the third by 5. The output of these nodes are merged together in an ascending order. The program can be represented by the opposite dataflow





graph, and can be written in Luswim(N)

as:

```
ham235
  where
    ham235 = 1 fby merge(2*ham235,merge(3*ham235,5*ham235));
    merge(x,y) = small
      where
        small = if p then xx else yy fi;
        xx = x upon xx<=yy;
        yy = y upon xx>=yy;
        p = xx < yy;
      end;
  end
```

### 1.3- Nested Iteration in Lucid:

We have mentioned earlier that Lucid is Luswim with **currenting** or **nested iteration**. In this section, we give a general description of such a concept. For more details, the reader is referred to [AsWa79, AsWa80, WaAs84].

To illustrate the concept of 'currenting' or 'nested iteration' in Lucid we give here a simple example in Lucid(N). Suppose we want to write a program which takes two streams of input values and returns the stream consisting of raising each component of the first to the power of the corresponding component of the second. That is, if the input sequences are

<2, 3, 5, 6, ... >

<0, 2, 3, 2, ... >

then the result sequence is <2<sup>0</sup>, 3<sup>2</sup>, 5<sup>3</sup>, 6<sup>2</sup>,

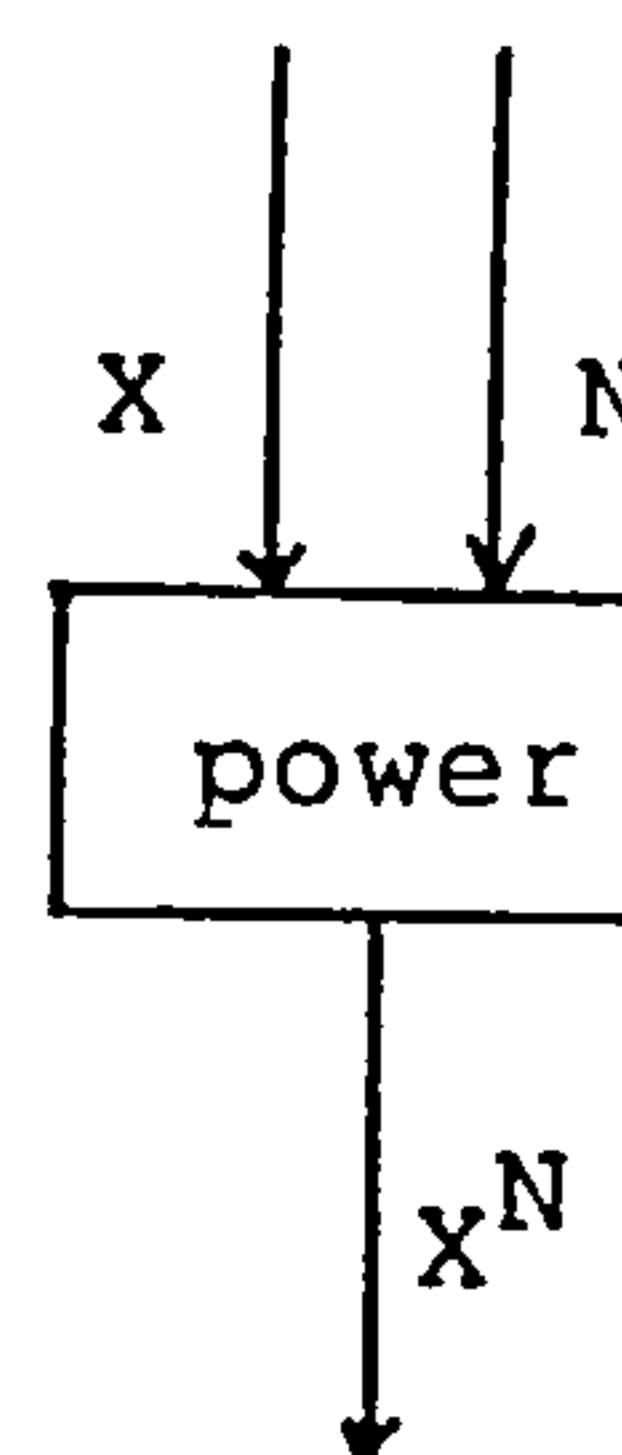
that is <1, 9, 125, 36, ... >

Of course, we can write such a program in Luswim(N) using recursion as

```
power(x,n) where
  power (a,b) = if b eq 0 then 1
                else a * power(a,b-1) fi;
end
```

however, the idea here is to use iteration only.

Informally speaking, we want to define a box which takes two sequences  $X$  and  $N$ . Then, multiply each component  $X_m$  in  $X$  by itself  $N_m$  times. A naive way of writing the program is



```

result where
  I = 0 fby I+1;
  P = 1 fby P*X;
  result = P as I eq N;
end
  
```

However, if we interpret the above as a `Luswim(N)` program then the result, if there is to be any, is not correct. The reason is, the flow of the input sequences  $X$  and  $N$  are (informally) synchronized with the values of the expressions inside the program. That is, if  $X$  takes the value

$\langle 2, 3, 5, 1, 2, \dots \rangle$

then, as  $P = 1 \text{ fby } P * X$ , the value of  $P$  will be the running product of the sequence  $X$ ; i.e

$P = \langle 1, 2, 6, 30, 30, 60, \dots \rangle$

Moreover, as the flow of the input sequence  $N$  is running simultaneously with the expressions inside the box,  $I$  in the definition of `result` is looking for a value of  $N$  which may never show up. That is, the program may never terminate let alone be correct.

If we represent the above program as a box, then we need a 'gate' at each input port of the box. These gates allow one value of  $X$  and  $N$  at-a-time. Once a value of  $X$  and a value of  $N$  are passed into the box, the flow of  $X$  and  $N$  (to the box) is frozen and a subcomputation is triggered inside using the current (or the frozen) values of  $X$  and  $N$ . When the box produces a result the gates open and let the next components of  $X$  and  $N$  in, and so on. In Lucid, we represent such an operational concept by the construct `current`. The syntax of `current` is

*var is current expression*

The occurrences of variable symbols in *expression* are all globals to the currented clause, and the occurrence of *var* is local. Thus, in the above example we add to the where clause

**X is current X**

and **N is current N**

These statements, informally, mean that the value of the occurrence of X (respectively N) inside the clause is the current value of the occurrence of X (respectively N) outside. Instead of using the same variable symbols X and N to represent the frozen values, we can introduce completely new variable symbols, say x and n. The above program can be written then as:

**result where**

**x is current X;**

**n is current N;**

**I= 0 fby I+1;**

**P= 1 fby P\*x;**

**result= P asa I eq n;**

**end**

To cast further light on the concept of 'currenting' or 'freezing' and relative time in Lucid, we shall embed the above program as a where-clause in another program. Let us assume that the variables X and N are defined in an outer main clause, that is



power where

$X = 1 \text{ fby } X+2;$

$N = 0 \text{ fby } N+2;$

power = result where

x is current X;

n is current N;

$I = 0 \text{ fby } I+1;$

$P = 1 \text{ fby } P \cdot x;$

result = P as I eq n;

end;

end

In the main where clause of the program, X and N are synchronized with each other; we can think of that as being computed simultaneously. They are synchronized also with the value of result which is the value of the whole inner where-clause. That is, for any time t

$$\begin{aligned} \text{result}(t) &= X^N(t) \\ &= X(t)^{N(t)} \end{aligned}$$

However, for each instance in time in the outer where-clause the inner clause is triggered and starts, from nought, its own sequence of time while the outer one is still. Thus, for every instance of X and N, the value I starts the count from 0. Also, for every instance of X, say 3, P is the sequence of powers of 3

$$\langle 1, 3, 9, 27, 81, \dots \rangle$$

The concept of "currenting" or "freezing" in Lucid corresponds, exactly, to nested iteration in conventional languages where we freeze certain values and let some others run till we extract the required result. For example, in the above program the variable symbols take the following values:

time	0	1	2	...
X	1	3	5	...
N	0	2	4	...

x	<1,1,1,...>	<3,3,3,...>	<5,5,5,...>	...
n	<0,0,0,...>	<2,2,2,...>	<4,4,4,...>	...
I	<0,1,2,...>	<0,1,2,...>	<0,1,2,...>	...
P	<1,1,1,...>	<2,2,2,...>	<4,4,4,...>	...
result	<1,1,1,...>	<9,9,9,...>	<625,625,...>	...

subcomputation

power	1	9	625	...
-------	---	---	-----	-----

From the above interpretation of currented where-clauses, we see that their semantics is completely different from those of Luswim. A clause in Luswim defines an iterative process, or computation, which runs in synchrony with the equations of the clause containing it. A currented where-clause, on the other hand, defines an infinite sequence of computations. Each of these computations proceeds while the computation of the outer clause is still. Hence, for each instance of time in the outer computation we have a new sequence of time for the subcomputation of the inner one. See for example, I in the power program.

It is worth mentioning here that the value of a global variable symbol inside a currented where clause is the sequence value which such a variable symbol takes in the clause it is defined in. In other words, the definition of such a variable symbol can be imported to the clause without affecting the semantics of the program. For example, in the power program above, we can rewrite the program by defining I in the outer clause without affecting the semantics of the program; i.e. the program can be written as:

```

power where
  X = 2 fby X+3 ;
  N = 1 fby N+1;
  I = 0 fby I+1;
  power = result where
    x is current X;
    n is current N;
    P = 1 fby P*x;
    result = P asa I eq n ;
  end;
end

```

#### 1.4- Examples:

We give here a program for computing the first 50 prime numbers. The program is written in pLucid which is a member of the Lucid family developed and implemented at the University of Warwick [FMY83, WaAs84]. It is the member Lucid(P), where P is the algebra over the data types: lists, words, strings, integers, reals and the types error and eod (end of data). The definition of the pLucid syntax using the BNF formalism appears in appendix A.

```

if index < 99 then p else eod fi

where

  index = 0 fby index+1;

  p = 2 fby nxprime(p);

  nxprime(n) = q asa isprime(q)

    where

      N is current n;

      q = N+1 fby q+1;

    end;

  isprime(n) = p*p>N asa condition

    where

      N is current n;

      condition = p*p >= N or N mod p eq 0;

    end;

end

```



### 1.5- Survey of Lucid Implementations:

Since the invention of Lucid "around 1972", a considerable amount of research has been carried out on studying ways of implementing the language. The outcome of such research can be classified according to two main aspects; (1) the model of evaluation used (such as dataflow, message passing, or conventional compilation), and (2) the version of Lucid which was implemented. Lucid has passed through four stages of development; each can be signified by a distinct version. In this section, we shall describe, briefly, the four versions of Lucid, and the research which was done on implementing each version. Later on, we shall stop for a while on two major works; the interpreter of C.Ostrum and that of A.Faustini. Beside being the only working 'physical' interpreters for full Lucid, this thesis was originally triggered by the basic ideas of those interpreters.

It is worth noting here that, apart from the lexical and syntactical analysis, all Lucid interpreters incorporate a front end pre-processor. This front end produces an intermediate code which is basically a linear member of the Lucid family.

#### 1.5.1- Basic Lucid:

This is the oldest published version of the language [AsWa76, AsWa77a]. It lays the basic concepts underlying the Lucid formal system as a system for writing and reasoning about programs. A program in Basic Lucid is a linear (not structured) set of assertions (the order of the assertions in the program is irrelevant). The Lucid operations which were available in B.Lucid are first, next, fby, asa, whenever and the Lucid operator 'latest' for nested iteration. The assertions are considered as statements (in the program) as well as axioms and rules of inference for reasoning about the program.

## Implementing Basic Lucid:

1- D.May and W.Wadge : Probably, this is the first implementation of a member of Lucid (around 1974). The member of Lucid implemented was Warwick Lucid, which is Basic Lucid together with the operations of BCPL (including shift operators) and a collection of operations for manipulating arrays. The interpreter was written in BCPL and based on demand-driven tagged dataflow concepts (time and space parameters). Probably, it was the first attempt to evaluate Lucid by education on a conventional machine. The work, however, was never completed nor documented.

2- W.Wadge: An interpreter for Basic Lucid in FORTRAN around 1976. The scheme of interpretation was based on demand driven tagged dataflow and along the lines of May's interpreter. It was never completed nor documented.

3- T.Cargill: This is very similar to D.May's interpreter of Warwick Lucid [Carg76].

### 1.5.2- Clause Lucid:

The second version is what we will refer to, here, as Clause Lucid. Basic Lucid was extended to allow user defined functions and scope conventions using a collection of 'clauses'. 'Lucid clauses' are compound Lucid assertions [AsWa77b] and are of four types:

- the **compute** clause: for defining blocks,
- the **mapping** clause: for defining conventional pointwise functions,
- the **produce** clause: for defining blocks with memories (or processes),
- the **function** clause: for defining general kinds of coroutines.

### Implementing Clause Lucid:

1- P.Gardin: An implementation written in FORTRAN [Gar78]. Clause Lucid is transformed into Lucode (an intermediate Lucid code). In Lucode, Gardin incorporates some modal and intensional operators for binding globals and passing parameters. An interpreter was actually written but it was very slow and inefficient.

2- C.Hoffman: Compiled a subset of Basic Lucid into ALGOL60. However, he has done considerable work on the theoretical aspects and the operational semantics of Lucid clauses [Hof78, Hof80].

3- M.Farah: Gave a formal description for compiling a subset of Clause Lucid into an ALGOL-like sequential code [Far77].

4- V.Bush: Implemented a subset of Clause Lucid on the Manchester dataflow machine [Bus79].

### 1.5.3- Structured Lucid:

All the clauses of the previous version were abandoned and merged into one clause; the **valof** [AsWa80, AsWa79]. A **valof** clause is a set of Lucid assertions (definitions). One of these assertions defines the variable symbol **result**. Global variable symbols in a **valof** clause are either *elementary* or *non-elementary*. Non-elementary globals are those which are called (in the present version) **currentted** (or frozen) globals.

### Implementing Structured Lucid:

1- B.Finch [Fin81]: Studied the operational views of translating Structured Lucid into message passing actors.

2- C.Ostrum [Ost81]: We shall give a general description of this interpreter in section 1.5.5.



3- J.Sargeant [Sar82]: Implemented the language on the Manchester dataflow machine.

4- C.Denbaum: Compiling Clause Lucid into ANPL (a non-procedural language) which is a subfamily of Lucid. ANPL is then compiled into ACL which is a PASCAL-based programming language with message passing coroutines [Denb83].

#### 1.5.4- Book Lucid:

Which is the latest version of the language and is fully described in the book [WaAs84] and briefly in this thesis. The ISWIM where clause was incorporated in the language instead of the `valof`, together with the concept of currenting or freezing.

#### Implementing Book Lucid:

1- P.Pilgram: Translating Book Lucid into message passing actors on a conventional machine. The language is compiled into LUX which is a PASCAL-like language with message passing actors [Pil83].

2- A.Faustini: An implementation based on Ostrum's interpreter. We shall give a general description of this interpreter in section 1.5.6.

#### 1.5.5- Ostrum's Interpreter:

This was an implementation for Luthid (Lucid with Lisp); a member of Structured Lucid whose algebra of data types was over finite S-expressions with numeric and symbolic atoms. The interpreter was written in C and YACC, and is running under UNIX. The interpreter was not efficient in terms of speed and storage allocation; but it gained a considerable amount of significance due to

the way in which computation was handled.

The interpreter is divided into two main processors; the `comp` processor and the `run` processor. `Comp` takes the source program and produces the intermediate code (called the `e-code`) which is in the form of expression trees. Each variable in the program is represented by a number. If the variable is local, the number represents the position in the expression tree table in which the expression defining the variable can be found. If the variable is a formal parameter the number is the position of such a parameter in the formal parameter list of the function.

`Run` interprets the `e-code` produced by `comp` using a demand-driven tagged strategy. At any given time in the evaluation we have two separate stacks of environments, a time-stack and a place-stack. Whenever we start evaluating a nested environment, the top environment in the stack is suspended and a new environment is pushed on top of the time-stack. The place-stack is for function calls. In evaluating a function call, the top place-stack is suspended and a new environment is pushed on top of the place-stack. The place-stack is also updated when entering a currented `valof` (a clause with non-elementary globals).

At any moment of time and place, when calling a local variable such a variable is requested at the same time and place (the present environment). If the variable is an elementary global the place-stack is popped until we reach the place where such a variable is defined. If it is a non-elementary global, the place-stack is popped to the defining environment and the time-stack is popped to the latest environment.

When requesting a function call, a display entry of three components is created; one component contains the place-environment in which the function is defined, another describes the place of the parameters, and the third

describes the time of the call if the function is frozen. If the function is not frozen then the third component will be left empty.

We should notice here that both the place-environments and the time-environments are represented by numbers. A place-environment is represented by the index of the function (which pushed the environment) in the expressions table. The time-environment is represented by the time at which the environment was pushed on top of the stack. Therefore, we end up evaluating the expressions of the intermediate code tagged with two sequences of numbers; the first representing the time-stack and the other representing the place-stack. Thus, a demand-driven tagged evaluation.

#### 1.5.6- Faustini's Interpreter:

This is an implementation of the language pLucid [WaAs84, FMY83] which is a member of Book Lucid. The implementation is written in C and YACC and is running on UNIX. At present, it is the most efficient and generally used implementation of Lucid.

The interpreter of this implementation is basically that of C.Ostrum. However, many major developments and refinements have been done on Ostrum's interpreter. These refinements are:

- 1- Incorporating the data types of POP-2; lists, strings, words, integers, reals and the types error and eod (for end of data). Furthermore, better run-time error handling facilities were added to the interpreter.
- 2- The front end processor (Ostrum's `comp`) was changed to handle the `where` clause of Book Lucid and the `is current` assertion. New and better facilities for syntactic error messages were added.
- 3- A heuristic procedure for garbage collection was implemented with the system which dramatically increased the efficiency of the interpreter.



4- The language pLucid enables programmers to incorporate UNIX commands in their programs. Therefore, while a pLucid program can be considered as a UNIX filter, a UNIX command can be incorporated in a pLucid program as a processing filter. The interpreter handles such filters efficiently and thus creates a compatible environment for programming with UNIX.

## Chapter 2

# Intensional Logic and Intensional Algebras

### 2.0- Introduction, Semantics vs Pragmatics:

Formal linguistics is the intersection of philosophy, logic and linguistics and is concerned with the study of the syntax and semantics of languages. Being the object of this study, the abstract notion of a language has a great importance as it is the bridge linking formal languages, used by logicians, with natural languages.

While syntax is concerned with the formation of expressions and their linguistic structure, semantics is concerned with the relation between expressions and their meanings, in the every day use of the word 'meaning'. Both logicians and formal linguists recognize two theories in the study of meaning of languages, the theory of reference, and the theory of meaning or pragmatics, where the word 'meaning' here is technical. It is worth noting here, however, that some logicians identify the word 'semantics' with the theory of reference developed by Tarski, Goedel, Hilbert, and others. This was later known as the model theoretic approach to semantics [Tar56], but here we shall reserve the word 'semantics' for the study of meaning in contrast with 'syntax' as the study of the purely formal structure of the language.

The theory of reference or denotation is concerned, as we shall see later, with the relation between expressions and the objects they denote, or refer to, according to a prior given denotation function. On the other hand, 'Pragmatics' is the study of the relation between expressions, the objects they denote, and

the contexts of use or utterance. In fact, we can say that the theory of pragmatics is a generalization of the theory of reference.

In Tarski's model theoretic approach, the semantic procedure consists of supplying a 'model' consisting of a domain of discourse together with a function assigning meaning or 'denotation' to the constant symbols in the language. Hence, a semantics for a language  $L$  is given by a model  $\langle D, F \rangle$ , where  $D$  is a set of objects and  $F$  is a function, the denotation function. The function  $F$  assigns to each  $n$ -ary operator symbol in the signature of  $L$  an operation of degree  $n$  over the set  $D$ , and for each  $n$ -ary predicate symbol in the signature of  $L$  an  $n$ -place relation over  $D$ .

An environment, in this approach, is simply a function which assigns to every  $n$ -ary variable symbol a function over the domain of discourse. That is, a function in  $[(D)^n \rightarrow D]$ . Thus an environment is a function in  $[V \rightarrow [D^n \rightarrow D]]$  where  $n$  is a natural number and  $V$  is the set of variable symbols.

Given a model together with an environment, more complex syntactic entities can be evaluated systematically using a set of semantic rules. It is worth mentioning here that an essential feature is Frege's principle of modularity, that the denotation of the whole should be a function of the denotations of the parts.

Montague's pragmatic theory is concerned with the study of **indexical expressions**; that is "words and sentences of which the reference cannot be determined without knowledge of the context of use" [Mon74]. For instance, the object which the pronoun 'I' denotes depends on the context of use which determines the speaker to whom the pronoun 'I' refers. Similarly, the proposition represented by the sentence 'Socrates is dead' depends on the context which determines the time at which the sentence was uttered.



The contrast between these two approaches to the study of meaning of languages can be shown in the following example. Let  $S$  be the expression represented by the sentence

**the president of USA**

The meaning (the value) of the expression above in the model theoretic approach (the denotation of the expression) is the person who is regarded as the president of the USA. It is a single value in the domain of discourse.

On the other hand, the value of the expression above in pragmatics (the intension of the expression) is the set of all past, present and future presidents of the USA. Notice that, we talked about the intension, taking moments of time as the main dimension of the various contexts of interpretation. This is just for the sake of simplicity in this particular example. We shall see, in later sections, that the complexity of contexts is dependent on the language to be interpreted. Thus, given a set of contexts together with a domain of discourse, the pragmatic value of an expression (the intension) is a function which assigns a single value in the domain to each context. For example, if we consider the set of contexts to be the set of years (represented by  $\omega$ ), then the intension of the proposition above is a function which maps each year (element in  $\omega$ ) to the person regarded as the president of USA in that particular year. We shall show in section 2.5 that these two notions of meaning are interdefinable.

## 2.1- Intensional Logic:

Intensional logic emerged from the theory of meaning and pragmatics to give interpretation to languages with intensional phenomena like time and place. These natural-like formal languages allow a logician to talk about entities such as pronouns, adverbs or demonstrative pronouns, whose values change with contexts.

The *intension* of an expression is an indexed family of values. Each member of this family is the value of the expression at a certain index. That is, the intension of an expression is a function which maps each index to the value of the expression at that index. When one thinks of the values of these expressions as being indexed by indices specifying the various complexities of contexts of use or utterance, the reason for the above terminology, i.e. indexical expressions, becomes clear.

For instance, the pronoun 'I' should be indexed with an index specifying the utterer. The intension of the pronoun 'I' is a function mapping the set of contexts of utterance to the 'person' who is regarded as the speaker in that context. The sentence 'John is a student', however, should be indexed with an index specifying the time of utterance. Thus, if we think of the set of indices being the set of years  $Y = \{1982, 1983, 1980, 1978\}$ , then the intension, relative to the set of indices  $Y$ , of the proposition  $S$ , represented by the sentence 'John is a student', is a function which maps each year to a truth value. If 'John' was a student in 1980 only, then the intension of the proposition  $S$ , relative to the set of indices  $Y$ , is the function

$$\text{Int}_Y(S) = \{ \langle 1983, \text{ff} \rangle, \langle 1982, \text{ff} \rangle, \langle 1980, \text{tt} \rangle, \langle 1981, \text{ff} \rangle \}.$$

To cast further light on intensional logic and pragmatics we define here a family  $L$  of intensional languages. These will be on the same line as those of Montague [Mon74], however, slightly different. Members of  $L$  are typeless and any mention of the word 'type' here will be taken as purely informal. Moreover, we shall consider  $L$ -formulas as  $L$ -terms. Hence, in the interpretation of the language we shall talk about values of terms rather than validity and satisfaction of formulas. This in some sense coincides with our intended use of the language as an implementation technique rather than a proof system. Moreover,  $L$  is a

family of first order intensional languages, i.e. we allow first order intensional operators only.



### 2.1.1- The Intensional Language $L(\Sigma)$

#### Syntax :

Let  $\Sigma$  be a set of constant symbols with different arities, and assume that we have an unlimited number of variable symbols with various arities.

The set of terms of  $L(\Sigma)$  is defined recursively as follows:

- If  $\psi$  is an  $n$ -ary constant symbol in the signature  $\Sigma$ ,

or an  $n$ -ary variable symbol, and

$\xi_0, \xi_1, \dots, \xi_{n-1}$  are  $n$  terms,

then

$\psi(\xi_0, \xi_1, \dots, \xi_{n-1})$  is a term.

### 2.1.2- Intensional Interpretation:

Our interpretation of intensional languages will be slightly different to that of Montague's as the languages of the family  $L$  are typeless. However, we adopt his convention to use the more general words *interpretation* and *structure*, rather than *model*, as the latter is often identified with denotational semantics. Before giving a formalism for the semantics of the family  $L$ , we give a brief general account of the constituents of an intensional interpretation structure.

### 2.1.2.i- Indices and Possible worlds:

We described a model theoretic structure as, briefly, a pair  $\langle D, F \rangle$  where  $D$  is a set, and  $F$  is the denotation function. The meaning of an expression will, then, be a single element in the set  $D$ . In intensional logic, however, the meaning of an expression is an indexed family of objects; each corresponds to the value of the expression at a certain index. Hence, if the set of all possible denotations or the domain of discourse is the set  $D$  then there is a subset of  $D$  assigned to each expression. This subset is the set of all possible denotations of that particular expression. The word 'possible' here is dependent on the choice of indices chosen in the structure. So if we let the set of indices be the set  $U$  then the intensional meaning of an expression is a function which maps each index (element of  $U$ ) to an element in the domain of discourse.

For example, consider the language  $\text{Lucid}(Z)$  where  $Z$  is the algebra of the integers with their usual mathematical symbols. The intensional value of the numeral 2 is the function  $\{ \langle i, 2 \rangle : i \in U \}$ . As the set of possible worlds in  $\text{Lucid}$  is the set of natural numbers  $\omega$ , the intensional value of 2 (the function  $\{ \langle i, 2 \rangle : i \in \omega \}$ ) is the sequence  $\langle 2, 2, 2, \dots \rangle$ . Notice that the value of 2 is the same in every world in the universe; this is because  $\text{Lucid}(Z)$  extends the operators of  $Z$  pointwise. Thus the values of the constant symbols in the signature of  $Z$ , say  $+$ ,  $-$ ,  $4$ ,  $-2$ ; do not change from a world to another in the universe. In contrast, the value of the  $\text{Lucid}(Z)$ -expression

**5 fby 10 fby 15**

varies from a possible world (an index) to another. The intension of the expression is the sequence  $\langle 5, 10, 15, 15, 15, \dots \rangle$ ; i.e. the value at index 0 is 5, at index 1 is 10, and at any index greater than 1 is 15.

We shall call the set of indices chosen in the structure, say  $U$  above, the set of possible worlds, the set  $D$  the extensional domain, and the set  ${}^U D$  the

intensional domain or the set of possible intensions.

It is worth mentioning here that if the set of indices is the set of natural numbers, representing instances of time, then the special case of intensional logic evolved, is an instance of logic of tense, or temporal logic.

Possible worlds can be more complex depending on the language to be interpreted. For example, for a tense language the set of possible worlds might be the set  $\omega$  representing time. This is sufficient to interpret sentences like **John is a student**. However, for sentences like **John is a student here**, the set of possible worlds should be

$$\{ \langle t, p \rangle \mid t \text{ is an instance of time \& } p \text{ is a place} \}$$

so that we can interpret the adverb **here**.

The set of possible worlds is often the cartesian product of a family of sets, each member of this family specifying a certain aspect of the context of use or utterance.

### 2.1.2.ii- The Intensional Interpretation function:

Secondly, we have to supply an intensional interpretation function, or the **intension** function, which assigns a possible intension to each symbol in the signature of the language.

In general, the **intension** function assigns to any n-ary constant symbol a function mapping each possible world to an operation of degree n on the intensions of its arguments. i.e. if U is the set of possible worlds and D is the extensional domain then the intension of an n-ary constant symbol is a function in

$$[ ({}^U D)^n \rightarrow ({}^U D) ]$$

Clearly, the **intension** of a nullary constant symbol is equivalent to an element



in the intensional domain  ${}^U D$ ; or equivalently, a function from the set of possible worlds to the extensional domain.

Essentially, the intensional meaning of a term is a function from the set of possible worlds to the extensional domain. However, the intension function assigns values only to the constant symbols of the language. This is sufficient to interpret expressions like the  $\text{Lucid}(Z)$ -expressions

$$3 + 5 \text{ eq } 8, \quad 1 \text{ fby } 5.$$

For expressions with variable symbols, like

$$X + 5 \quad \text{or} \quad X \text{ whenever } Z,$$

we need an **intensional environment** to assign meaning to the variable symbols occurring in the expression.

Given a set of possible worlds  $U$ , and an extensional domain  $D$ , an **intensional environment** is a function which assigns to every  $n$ -ary variable symbol a function in

$$[({}^U D)^n \rightarrow ({}^U D)]$$

That is, an intensional environment is an element in the set of functions

$$[V \rightarrow \bigcup_{n \in \omega} [({}^U D)^n \rightarrow ({}^U D)]] \quad \text{where } V \text{ is the set of variable symbols}$$

Clearly, an intensional environment assigns to each nullary variable symbol an element in the intensional domain of the language, i.e. a function from the set of possible worlds to the extensional domain.

It must be emphasized that the intensional languages we are going to use in this thesis are all of zero order (nullary order intensional languages). Hence, it is sufficient to consider nullary intensional environments only. That is, intensional environments which assign values to nullary variable symbols only.

We give next a formal definition of an **intensional structure** in which we

define the **intension** function relative to the set of possible worlds of the structure. Then we define the **meaning** function relative to an intensional structure together with an intensional environment. It is worth noting here that an intensional environment must be defined relative to a certain structure so that it maps each variable symbol to an element of the intensional domain of that particular structure.

**Definition 1:**

Let  $\Sigma$  be a set of constant symbols with various arities,

and let  $L(\Sigma)$  be an intensional language.

An **intensional**  $\Sigma$ -interpretation, or a **structure**, for  $L(\Sigma)$  is a triple  $\langle U, D, F \rangle$  where

$U$  is a nonempty set of possible worlds,

$D$  is a nonempty domain of objects, the extensional domain,

$F$  is the intension function which assigns to every

$n$ -ary constant symbol  $\psi$  in the signature  $\Sigma$  a

function in  $[(^U D)^n \rightarrow ^U D]$

**Definition 2:**

Let  $S$  ( $=\langle U, D, F \rangle$ ) be an intensional structure.

An  $S$ -intensional environment  $\varepsilon$  is a function in

$[V \rightarrow \bigcup_{n \in \omega} [(^U D)^n \rightarrow ^U D]]$  where  $V$  is the set of variable symbols.

Hence, an intensional environment assigns to each nullary variable symbol a function in  $^U D$

**Definition 3:**

Let  $S$  ( $= \langle U, D, F \rangle$ ) be an intensional structure for  $L(\Sigma)$ .

Let  $\varepsilon$  be an  $S$ -intensional environment.

Then the **intension** of an  $L(\Sigma)$ -term  $\tau$  relative to the structure  $S$  and the  $S$ -intensional environment  $\varepsilon$  (in symbols  $\models_{S,\varepsilon}(\tau)$ ) is defined recursively as follows:

If  $\tau$  is of the form  $\psi(\xi_0, \xi_1, \dots, \xi_{n-1})$

where  $\psi$  is an  $n$ -ary constant symbol and

$\xi_0, \xi_1, \dots, \xi_{n-1}$  are  $n$  terms then

$$(\models_{S,\varepsilon} \tau) = F(\psi)(\models_{S,\varepsilon} \xi_0, \dots, \models_{S,\varepsilon} \xi_{n-1})$$

If  $\tau$  is a nullary variable symbol, then

$$(\models_{S,\varepsilon} \tau) = \varepsilon(\tau).$$

As we mentioned earlier, our interest is mainly with nullary intensional languages and nullary intensional environments. The above definition of the meaning function can be generalized, however, to handle first order intensional languages. Simply, by adding

If  $\tau$  is of the form  $\vartheta(\xi_0, \xi_1, \dots, \xi_{n-1})$

$\vartheta$  is an  $n$ -ary variable symbol and

$\xi_0, \xi_1, \dots, \xi_{n-1}$  are  $n$  terms then

$$(\models_{S,\varepsilon} \tau) = \varepsilon(\vartheta)(\models_{S,\varepsilon} \xi_0, \dots, \models_{S,\varepsilon} \xi_{n-1})$$



## 2.2- Intensional Algebras:

Before introducing the concept of intensional algebras, we shall recall some definitions of classical algebras- or what we shall call from now on extensional algebras. For more details on these algebras we refer the reader to [GTW78].

### Definitions 1:

Let  $\Sigma$  be a signature (a set of constant symbols).

An extensional  $\Sigma$ -algebra  $A$  is an ordered pair  $\langle F, D \rangle$  such that;  $D$  is a non-empty domain, and  $F$  is a function mapping each  $n$ -ary constant symbol in  $\Sigma$  to an operation of degree  $n$  on the domain  $D$ ; i.e. a function in  $[D^n \rightarrow D]$

### Notation :

If  $A (= \langle F, D \rangle)$  is an extensional  $\Sigma$ -algebra, then  $D$  is called the domain of  $A$  and denoted by  $|A|$ .  $\Sigma$  is called the signature of  $A$ .

A domain is a CPO which, at least, has the truth values  $\{tt, ff\}$  and Scott's undefined element  $\perp$ .

### Definition 2:

An intensional  $\Sigma$ -algebra  $B$  is simply an intensional  $\Sigma$ - structure. That is, it is a triple  $\langle U, F, D \rangle$  where

$D$  is a domain,

$U$  is a non-empty set;

and  $F$  is a function mapping each  $n$ -ary constant symbol in  $\Sigma$  to a function in  $[({}^U D)^n \rightarrow {}^U D]$ .

### Notation:

If  $A (= \langle U, F, D \rangle)$  is an intensional  $\Sigma$ -algebra, then

$U$  is the universe of  $A$ ,

$F$  is the intension function,

$D$  is called the **extensional domain** of  $A$ ,

and the set of all functions from the universe  $U$  to the extensional domain  $D$ , denoted by  ${}^U D$ , is called the **intensional domain** of  $A$  and denoted by  $|A|$ . We can say, then, that an intensional  $\Sigma$ -algebra  $A$  maps each  $n$ -ary operator symbol in the signature  $\Sigma$  to a function of degree  $n$  on its intensional domain. Henceforth, when we talk about the domain of an intensional algebra we mean the intensional domain.

**Definition 3:**

For any two algebras  $A$  and  $B$ ,  $A$  is said to be a **restriction** of  $B$ , or equivalently  $B$  is an **expansion** of  $A$ , (in symbols  $A \subset B$ ) if

$$B / \text{signature}(A) = A$$

That is, the signature of  $A$  is a subset of the signature of  $B$  and the algebra  $B$  assigns to the constant symbols of the signature of  $A$  the same functions as those assigned by  $A$ . In other words, for every constant symbol  $\psi$  in the signature of  $A$ ,

$$\psi_B = \psi_A$$

**Definition 4:**

Let  $A$  and  $B$  be two intensional  $\Sigma$ -algebras with the same universe. A  $\Sigma$ -homomorphism  $f: A \rightarrow B$  is a function such that for any  $n$ -ary operation symbol  $\psi$  in the signature of  $A$  and any  $a_0, \dots, a_{n-1}$  in  $|A|$

$$f(A(\psi))(a_0, \dots, a_{n-1}) = (B(\psi))(f(a_0), \dots, f(a_{n-1}))$$

**Definition 5:**

Let  $I_\Sigma$  be the identity function for composition in the class of  $\Sigma$ -homomorphisms. That is, for any  $\Sigma$ -homomorphism  $f$

$$f \circ I_\Sigma = I_\Sigma \circ f = f$$

Then  $g$  is called the **inverse** of  $f$ , and denoted by  $f^{-1}$ , if and only if

$$g \circ f = f \circ g = I_\Sigma$$

**Definition 6:**

Let  $h$  be a  $\Sigma$ -homomorphism. Then  $h$  is called a  $\Sigma$ -isomorphism iff there is a  $\Sigma$ -homomorphism  $g$  such that  $g = h^{-1}$ .

Any two  $\Sigma$ -algebras  $A$  and  $B$  are isomorphic, written as  $A \equiv B$  iff there exist a  $\Sigma$ -isomorphism from  $A$  to  $B$ .

**Definition 7:**

Let  $A$  be an extensional  $\Sigma$ -algebra, and  $U$  be a set. The pointwise extension of  $A$  upon  $U$  is the intensional  $\Sigma$ -algebra (in symbols  $A^U$ ) defined as follows: for every  $n$ -ary operator symbol  $\vartheta$  in the signature  $\Sigma$ ,  $A^U$  assigns a function of degree  $n$  on the set  $^U|A|$  such that

$$\forall i \in U, \text{ and } \forall \xi_0, \dots, \xi_{n-1} \text{ in } ^U|A|$$

$$A^U(\vartheta(\xi_0, \dots, \xi_{n-1}))(i) = A(\vartheta(\xi_0(i), \dots, \xi_{n-1}(i)))$$

### 2.3- The Family of Intensional Algebras $\mathcal{L}\mathcal{U}$

The language Basic Lucid [AsWa80] is a linear equational language in which the value of an expression is a sequence (or an  $\omega$ -history). Such a sequence or a history can be thought of as representing the values which the expression takes as time changes. For example, in the language  $\text{BLucid}(Z)^\dagger$  where  $Z$  is the algebra of the integers represented by their usual symbols; the value of the expression  $3$  is the sequence  $\langle 3, 3, 3, \dots \rangle$ ,

and the value of the expression  $1 \text{ fby } 3 \text{ fby } 5$  is the sequence

$$\langle 1, 3, 5, 5, 5, 5, \dots \rangle.$$

Thus if we consider the set of natural numbers  $\omega$  as being the universe of possible world, then we can consider the algebra  $\mathcal{L}\mathcal{U}(Z)$  as an intensional

---

$\dagger$  Basic Lucid

algebra whose universe of possible worlds is the set  $\omega$ .

The algebraic function  $Lu$  then is a function which maps extensional algebras, like  $Z$  in the above example, to intensional ones, like  $Lu(Z)$ . It defines a family of intensional algebras in which each member is uniquely determined by the extensional algebra it is applied to. That is, it defines the family

$$\{Lu(A): A \text{ is an extensional algebra}\}.$$

Given an extensional  $\Sigma$ -algebra  $A$ ,  $Lu(A)$  is the intensional  $\Sigma'$ -algebra which extends  $A$  pointwise over the universe  $\omega$  and assigns meaning to the symbols

**first, next, fby, whenever, asa, upon, attime**

Clearly, the signature of the algebra  $Lu(A)$  is bigger than that of  $A$ , for any  $A$ . It is the union of the signature of  $A$  together with the Lucid operator symbols. However, as the signature of  $Lu(A)$  is a function of (uniquely determined by)  $\Sigma$ , we say that  $Lu(A)$  is an intensional  $\Sigma$ -algebra. We define here the algebraic function  $Lu$  which defines the family of intensional algebras

$$\{Lu(A): A \text{ is an extensional algebra}\}.$$

**Definition 8:**

Let  $A (= \langle F, D \rangle)$  be an extensional  $\Sigma$ -algebra. Then  $Lu(A)$  is the least intensional  $\Sigma'$ -algebra whose universe of possible worlds is the set of natural numbers  $\omega$ , and whose signature  $\Sigma'$  is the set

$$\Sigma \cup \{\text{first, next, fby, whenever, asa, upon, attime}\}$$

such that:

(a):  $Lu(A)$  extends  $A$  pointwise upon the universe  $\omega$ .

That is, for every  $n$ -ary constant symbol  $\psi$  in  $\Sigma$ ,

for every  $n$  sequences  $a_0, \dots, a_{n-1}$  in  ${}^\omega D$ ,

and for every  $i \in \omega$

$$(Lu(A)(\psi)(a_0, \dots, a_{n-1}))_i = F(\psi)(a_0(i), \dots, a_{n-1}(i))$$



(b):  $Lu(A)$  assigns meaning to the Lucid operation symbols as follows

For every  $x, y, t$  in  ${}^{\omega}D$ , and for every  $i \in \omega$

$$(Lu(A)(first(x)))_i = x_0$$

$$(Lu(A)(next(x)))_i = x_{i+1}$$

$$(Lu(A)(fby(x, y)))_i = \text{if } (i \text{ eq } 0) \text{ then } x_i \text{ else } y_{i-1}$$

$$(Lu(A)(wvr(x, t)))_i = \text{if } t_i \text{ then } x_i \text{ else } \perp$$

$$(Lu(A)(asa(x, t)))_i = \text{if } \exists j \leq i \ t_j \text{ and } (k \leq j \rightarrow \text{not } t_k) \\ \text{then } x_j \text{ else } \perp$$

$$(Lu(A)(upon(x, t)))_i = x_n \text{ where } n = \text{Card } \{j \leq i : t_j\}$$

and Card is the cardinality function on sets.

$$(Lu(A)(attime(x, t)))_i = x_{t_i}$$

## 2.4- Synchronic and Asynchronic Operators:

### Definition:

Let  $A(=\langle U, F, D \rangle)$  be an intensional  $\Sigma$ -algebra, and let  $\psi$  be an intensional operator over  $A$ . Then  $\psi$  is called **pointwise** iff

$$\forall i, j \in U, \forall X, Y \in {}^U D, X(i) = Y(j) \implies \psi(X)(i) = \psi(Y)(j)$$

### Definition:

Let  $A(=\langle U, F, D \rangle)$  be an intensional algebra. An  $n$ -ary operation  $\psi$  in  $A$  is called **synchronic** iff

$$\forall i \in U, \forall X, Y \in {}^U D, X(i) = Y(i) \implies \psi(X)(i) = \psi(Y)(i)$$

That is, an operator is **pointwise** if and only if its value at a certain index is dependent on the values of its arguments at that same index. Hence, the meaning of a pointwise operator is the same at any index in the universe. For example, in the algebra  $\mathcal{L}\mathcal{U}(N)$

$$\forall i, j \in \omega \ ((\mathcal{L}\mathcal{U}(N)+)_i = (\mathcal{L}\mathcal{U}(N)+)_j).$$

Hence if the value of  $X$  is  $\langle x_0, \dots, x_n, \dots \rangle$

and the value of  $Y$  is  $\langle y_0, \dots, y_n, \dots \rangle$ ,

then the value of  $X+Y$  is  $\langle x_0+y_0, \dots, x_n+y_n, \dots \rangle$ .

So is the meaning of  $-$ ,  $\text{div}$ , and so on. On the other hand, an intensional operator is **synchronic** if and only if its value at any index in the universe is a function of the values of its arguments at that same index together with the index itself. That is, the meaning of the operator varies from an index to another. Hence, its value at any index is a function of both the index and the values of its arguments at that same index.

We want to emphasize here that we do not mean the every day use of the word *synchronic*; that is, occurring at the same time. *Synchronic* in this thesis will mean occurring at the same world in the universe no matter what the

dimensions (or the factors) of the worlds are; such dimensions might be time, place, or pairs of time and place, ... etc.

Clearly from the definitions above, all pointwise operators are synchronic.

### Example

Let  $Z$  be the usual extensional algebra of the integers. Then the operators  $+$ ,  $-$ ,  $*$ ,  $\text{div}$  in  $\mathcal{L}\mathcal{U}(Z)$  are all pointwise because for any operator  $\psi$  in  $Z$  and for any two sequences  $X, Y$  in the intensional domain  ${}^\omega Z$  where  $Z$  is  $|Z|$  ( the domain of  $Z$ ) we have

$$(\psi_{\mathcal{L}\mathcal{U}(Z)}(X, Y))_i = \psi_Z(X_i, Y_i)$$

so for any indices  $i, j \in W$ ,

$$\psi_{\mathcal{L}\mathcal{U}(Z)}(X, Y)_i = \psi_{\mathcal{L}\mathcal{U}(Z)}(X, Y)_j$$

If we add to the above algebra  $\mathcal{L}\mathcal{U}(Z)$  the operator **moment** such that

$$\forall i \in \omega \forall X \mathcal{L}\mathcal{U}(Z)(\text{moment})(X)_i = X^i$$

then the operator  $\mathcal{L}\mathcal{U}(Z)(\text{moment})$  is synchronic but not pointwise as its value depends on the index as well as the argument.

Clearly, for any  $i, j \in \omega$

$$i \neq j \implies \text{moment}(X)_i \neq \text{moment}(X)_j$$

(where **moment** is  $\mathcal{L}\mathcal{U}(Z)(\text{moment})$  ).

We shall call the operators which are not synchronic **asynchronic** intensional operators. The value of an asynchronic operator at any index in the universe depends, not only on the index and the values of its arguments at that particular index, but also on the values of its arguments at different indices in the universe.

### Example

The Lucid operators **first**, **next**, **fbv** are asynchronic (not synchronic).

Obviously,

$$\text{next}(X)_i = \text{next}(Y)_i \Rightarrow X_i = Y_i$$

for example, let  $X$  be  $\langle 2, 3, 4, 6, 7, 7, 7, \dots \rangle$

and  $Y$  be  $\langle 5, 7, 4, 6, 7, 7, 7, \dots \rangle$

then  $\text{next}(X)_1 = \text{next}(Y)_1 = 4$

however  $X_1 = 3$ , and  $Y_1 = 7$

Moreover, if  $X$  is  $\langle 5, 10, 15, 20, \dots \rangle$

and  $Y$  is  $\langle 5, 6, 7, 8, \dots \rangle$

then for any  $i$ ,  $\text{first}(X)_i = \text{first}(Y)_i = 5$

however  $X_j \neq Y_j$  for any  $j$  except 0.

This is because the value of  $\text{next}(X)_i$  is dependent on  $X_{i+1}$  and not on  $X_i$ .

Similarly, the value of  $\text{first}(X)_i$  is  $X_0$ , whatever  $X_i$  is.



## 2.5- The Intension vs The Extension:

Basically, there are two ways of studying intensional algebras. The first, we start with an extensional algebra, say  $A$ , and a universe of indices, say  $U$ , then extend the operations of  $A$  upon the universe  $U$  yielding the pointwise intensional algebra  $A^U$ . This simple intensional algebra is then enriched by adding non-pointwise operations. This method is suitable for the study of formal intensional languages.

In our thesis, we shall carry out this method for defining the intensional algebras we are going to use for the compilation process. For example, in the algebra  $Lu(A)$ , defined above, the operations of  $A$  are extended pointwise upon the universe  $\omega$ , then enriched by adding the Lucid operators.

The second approach to studying intensional algebras is mainly used in the study of natural languages where the semantics has to conform to the regimentation of the syntax on which logicians have no control. In this approach, we construct the intensional algebra such that the intension of an expression is the collection of its extensions at the worlds of the universe.

Clearly, the notions of **intension** and **extension** are related. Given the intension function  $A$  whose universe of possible worlds is a set  $U$ , we can derive the extension of  $A$  at a world  $i \in U$ . It is simply the  $i$ 'th projection of the intension. This is formalized in the following definition,

### Definition

Let  $A (= \langle U, F, D \rangle)$  be an intensional  $\Sigma$ -algebra.

and let  $i \in U$

The **extension** of  $A$  at  $i$ , in symbols  $Ext_i(A)$ , is the extensional  $\Sigma$ -algebra  $F$  such that

for every  $n$ -ary operator symbol  $\vartheta$  in the signature  $\Sigma$ ,

$$Ext_i(A)(\vartheta) = F(\vartheta) =$$

$$\lambda \langle d_0, \dots, d_{n-1} \rangle \in D^n. A(\vartheta) (\lambda u \in U. d_0, \dots, \lambda u \in U. d_{n-1})_i$$

$$\begin{array}{ccc} D^n & \xrightarrow{\quad} & D \\ \downarrow \lambda u. d_n & & \uparrow i \\ ({}^U D)^n & \xrightarrow{\quad} & {}^U D \end{array}$$

From the definition then,  $Ext_i(A)$  of a nullary symbol is the  $i$ 'th value in the sequence equivalent to its intension (intensional value).

However, the extension at a certain index in the universe is not enough to derive the intension; except of course the point wise intensional algebra. In point wise intensional algebras, the meaning of a symbol is the same in every world in the universe. Thus, knowing  $Ext_i(A)$  for some  $i$  in the universe of  $A$  is enough to derive (or to recover) the intension  $A$ .

Also, the set  $\{Ext_i(A) : i \in U\}$  (the extension of  $A$  at all worlds in the universe) is not enough to derive  $A$ ; unless all the operations of  $A$  are synchronic. This is because the extension of a synchronic operator at a point  $i$ , is not necessarily equivalent to its extension at a different point  $j$ .

For example, define the operator moment in  $Lucid(N)$  such that

$$\text{for every sequence } x, \forall i \in \omega, \text{moment}(x)_i = x^i$$

$$\text{then } i \neq j \Rightarrow \text{moment}(x)_i \neq \text{moment}(x)_j$$

Therefore, if we want to derive (or to recover) the intensional meaning of a synchronic operator in  $A$  we need to know the extension of such an operator at every point in the universe. Thus, to derive  $A$ , we need to know

$$\{Ext_i(A) : i \in U\}$$

The asynchronous operators of  $A$  cannot be derived knowing the extension. This

is because the value of an asynchronous operator at a world in the universe depends not only on the values of its arguments at that particular world and the world itself but on the values of the argument at other worlds in the universe.

The conclusion here is that, if we are given an intensional algebra,  $A$  say, then we can derive the extension of  $A$  at all worlds in the universe of  $A$ ; that is  $\{Ext_i(A) : i \in U\}$  where  $U$  is the universe of  $A$ . However, given  $\{Ext_i(A) : i \in U\}$ , we cannot recover the intensional algebra  $A$  unless the operators of  $A$  are all synchronic.

## 2.6- Galaxies and Clusters for Intensional operators

In section 2.4, we have classified the operations over an intensional algebra into three classes; pointwise, synchronic and asynchronous. For example, in the algebra  $\mathcal{I}\mathcal{A}(N)$ ,

pointwise like the operations  $+, -, *$  and  $\text{div}$

synchronic like the operation  $\text{moment}$

asynchronous like  $\text{first}$ ,  $\text{fby}$  and  $\text{next}$ .

We defined the asynchronous operators as those whose values in a certain world depend on the values of their arguments at different worlds in the universe. For example, the value of  $\text{first}(X)_i$  for any  $i \in \omega$  is a function of the value of  $X_0$ . Also, the value of  $\text{next}(X)_i$  for  $i \in \omega$  is a function of  $X_{i+1}$ , and so on.

However, it is not always the case that the value of an operator, at a certain world, depends on the value of the arguments at one single world, as is the case with  $\text{first}$  and  $\text{next}$  in  $\mathcal{I}\mathcal{A}$ . Some intensional operators may depend on the values of their arguments at a subset of the universe.

For example, consider the operator  $\text{hitherto}$  over the algebra  $\mathcal{I}\mathcal{A}(N)$  where

$$\forall i \in \omega, \forall X \in {}^{\omega}N$$

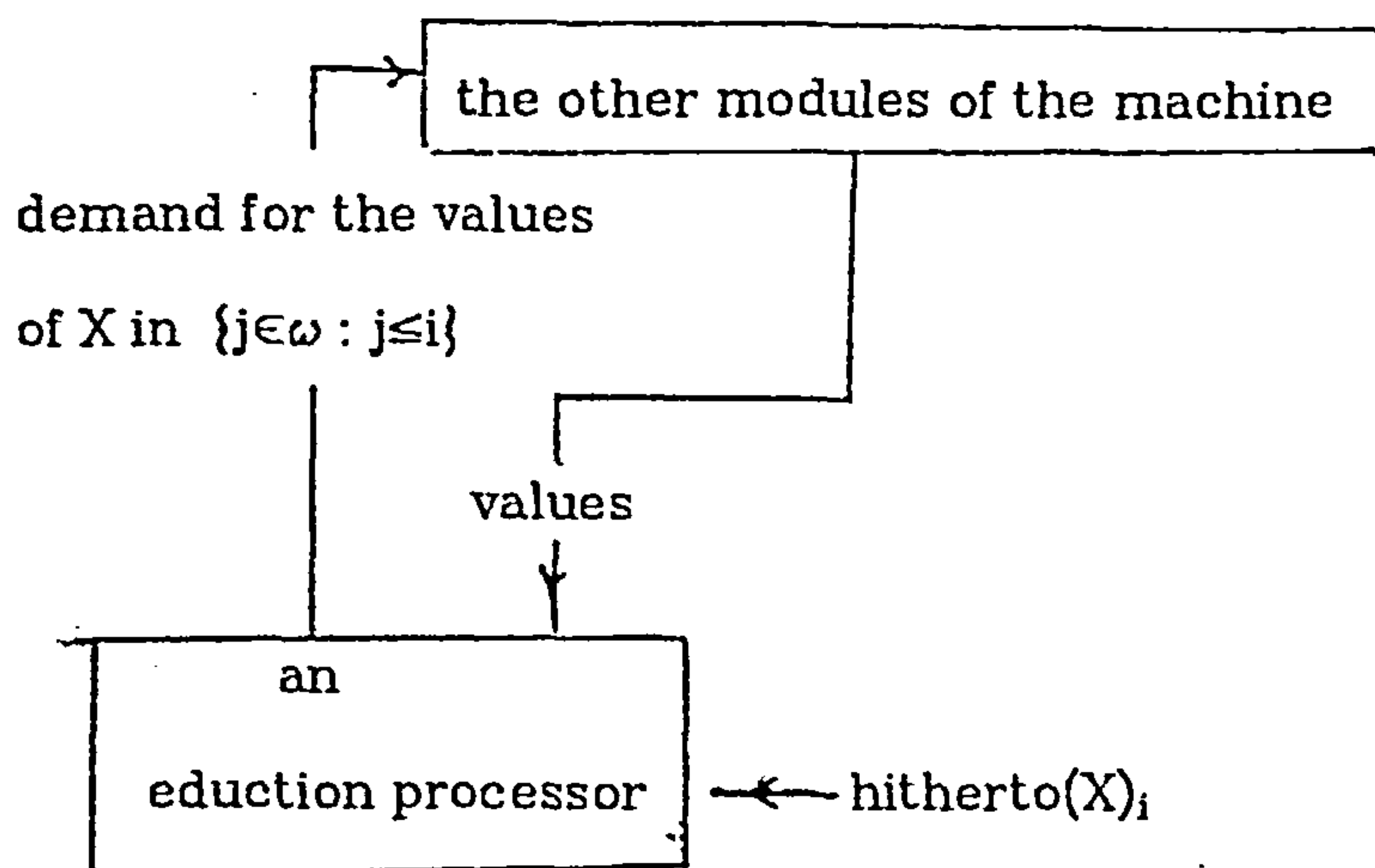
$$\text{hitherto}(X)_i = \begin{cases} \text{true} & \text{if } X_i \text{ and } \forall j < i (X_j) \\ \text{false} & \text{otherwise} \end{cases}$$

Clearly, the value of  $\text{hitherto}(X)_i$  is dependent on the value of  $X$  at all indices  $j$  where  $j \leq i$ .

In terms of education, this means that the value of  $\text{hitherto}(X)_i$ , for any argument  $X$  and any world  $i$ , cannot be determined unless we know the values of  $X$  at every world  $j$  less than  $i$ . Let us for the sake of simplicity consider a very naive view of the education machine as being a collection of education processors together with a token-value store. When an education processor, then, receives a demand for evaluating the expression  $\text{hitherto}(X)_i$ ; it responds by



demanding the values  $\{X_j : j \leq i\}$ . This can be illustrated as follows



In the case of  $\text{hitherto}(X)_i$ , the subset of the universe  $\{j \in \omega : j \leq i\}$ , is determined by (or a function of) the world  $i$ . However, there are operators whose values depend on the values of their arguments in a subset of the universe where such a subset is determined by the arguments themselves. For example, in the algebra  $\mathcal{L}\mathcal{U}(N)$

if  $X = \langle x_0, x_1, x_2, \dots, x_n, \dots \rangle$

and  $Y = \langle y_0, y_1, y_2, \dots, y_n, \dots \rangle$

then the value of  $(X \text{ asa } Y)$  at any world  $i \in \omega$  is the value of  $X$  in the least world  $j$  (under the natural ordering  $<_\omega$ ) where  $j \leq i$  and  $y_j$  is true. Thus, if

$Y = \langle \text{tt}, \text{ff}, \text{tt}, \text{ff}, \text{ff}, \dots \rangle$

then the value of  $(X \text{ asa } Y)$  in any world  $i$  is  $x_0$ .

If  $Y = \langle \text{ff}, \text{ff}, \text{ff}, \text{ff}, \text{ff}, \text{tt}, \dots \rangle$

then  $(X \text{ asa } Y)$  is  $\langle x_5, x_5, x_5, \dots \rangle$ .

Generally speaking, if  $\psi$  is an  $n$ -ary intensional operator, and  $j$  is an index in the universe, then for  $\langle x_0, \dots, x_{n-1} \rangle$  in the intensional domain, the value of  $\psi(x_0, \dots, x_{n-1})_j$  depends on a family of subsets of the universe. Each member of this family is associated with the operator, the index, and an operand  $x_i$ , for  $i \in n$ .

For example, for every  $i$  in  $\omega$  and for every sequence of values  $X$ , the value of  $\text{next}(X)_i$  depends on the value of the argument  $X$  in the set  $\{j : j = i+1\}$ .

The value of  $\text{fby}(X,Y)_i$  is dependent on a family of two sets of indices. The first is associated with  $\text{fby}$ ,  $i$ , and  $X$ ; which is the set  $\{0\}$ . The other is associated with  $\text{fby}$ ,  $i$ , and  $Y$ ; which is the set

$\{k : k=i-1\}$ . In other words  $\text{fby}(X,Y)_i$  is dependent on the set  $\{ \langle 0,k \rangle : k=i-1 \}$ .

We shall call the subsets of the universe associated with an operator  $\psi$ , a world  $i$ , and an argument  $X$  the **galaxy** of  $\psi$  at  $X$  and  $i$ . In symbols, we shall write  $\Delta(\psi,X,i)$ .

### Definition

Let  $B(=\langle U,F,D \rangle)$  be an intensional  $\Sigma$ -algebra.

For an  $n$ -ary operator  $\psi$  over  $B$ , any index  $i \in U$ , and  $X \in {}^U D$

the **galaxy** of  $\psi$  at  $X$  and  $i$ , in symbols  $\Delta(\psi,X,i)$ , is the set

$$\{J \subset U^n : \forall X' \in {}^U D, \forall j \in J \forall k \in n \ X_k(j_k) = X'_k(j_k) \Rightarrow \psi(X)_i = \psi(X')_i\}$$

Clearly, for a unary operator  $\vartheta$ ,  $i \in U$ , and  $X \in {}^U D$

$$\Delta(\vartheta,X,i) = \{J \subset U : \forall X' \in {}^U D \forall j \in J \ X(j) = X'(j) \Rightarrow \vartheta(X)_i = \vartheta(X')_i\}.$$

### Proposition:

Let  $B(=\langle U,F,D \rangle)$  be an intensional  $\Sigma$ -algebra.

Let  $\psi$  be an  $n$ -ary operator over  $B$ , then

$$\forall i \in U \forall \langle X_0, \dots, X_{n-1} \rangle \in ({}^U D)^n$$

$$J \in \Delta(\psi, \langle X_0, \dots, X_{n-1} \rangle, i) \Rightarrow \forall K \subset U^n (J \subset K \Rightarrow K \in \Delta(\psi, X, i))$$

Proof: Straight forward from the definition of  $\Delta$ .

Informally speaking, the proposition states the fact that if a set  $J$  is in the galaxy of  $(\psi, \langle X_0, \dots, X_{n-1} \rangle, i)$ ; then any subset of the universe which contains the set  $J$  is also in the galaxy  $\Delta(\psi, \langle X_0, \dots, X_{n-1} \rangle, i)$ .

**Examples:**

In the algebra  $\mathcal{L}\mathcal{U}(A)$

$$\forall X \forall i \Delta(\text{first}, X, i) = \{J \subset \omega : 0 \in J\}$$

$$\forall X \forall i \Delta(\text{next}, X, i) = \{J \subset \omega : i+1 \in J\}$$

$$\forall X \forall Y \forall i \Delta(\text{fby}, \langle X, Y \rangle, i) = \{J \subset \omega \times \omega : \langle 0, i-1 \rangle \in J\}$$

$$\forall X \forall i \Delta(\text{hitherto}, X, i) = \{J \subset \omega : (\forall j) j \leq i \Rightarrow j \in J\}.$$

Given an operator  $\psi$ , the galaxy of  $\psi$  at a world  $i$  determines how much information we need to know about the argument in order to compute the  $i$ 'th value of  $\psi$ . For example, given the Lucid(N)-expressions  $X$  and  $Y$ , the galaxy

$$\Delta(\text{fby}, \langle X, Y \rangle, i) = \{J \subset \omega \times \omega : \langle 0, i-1 \rangle \in J\}$$

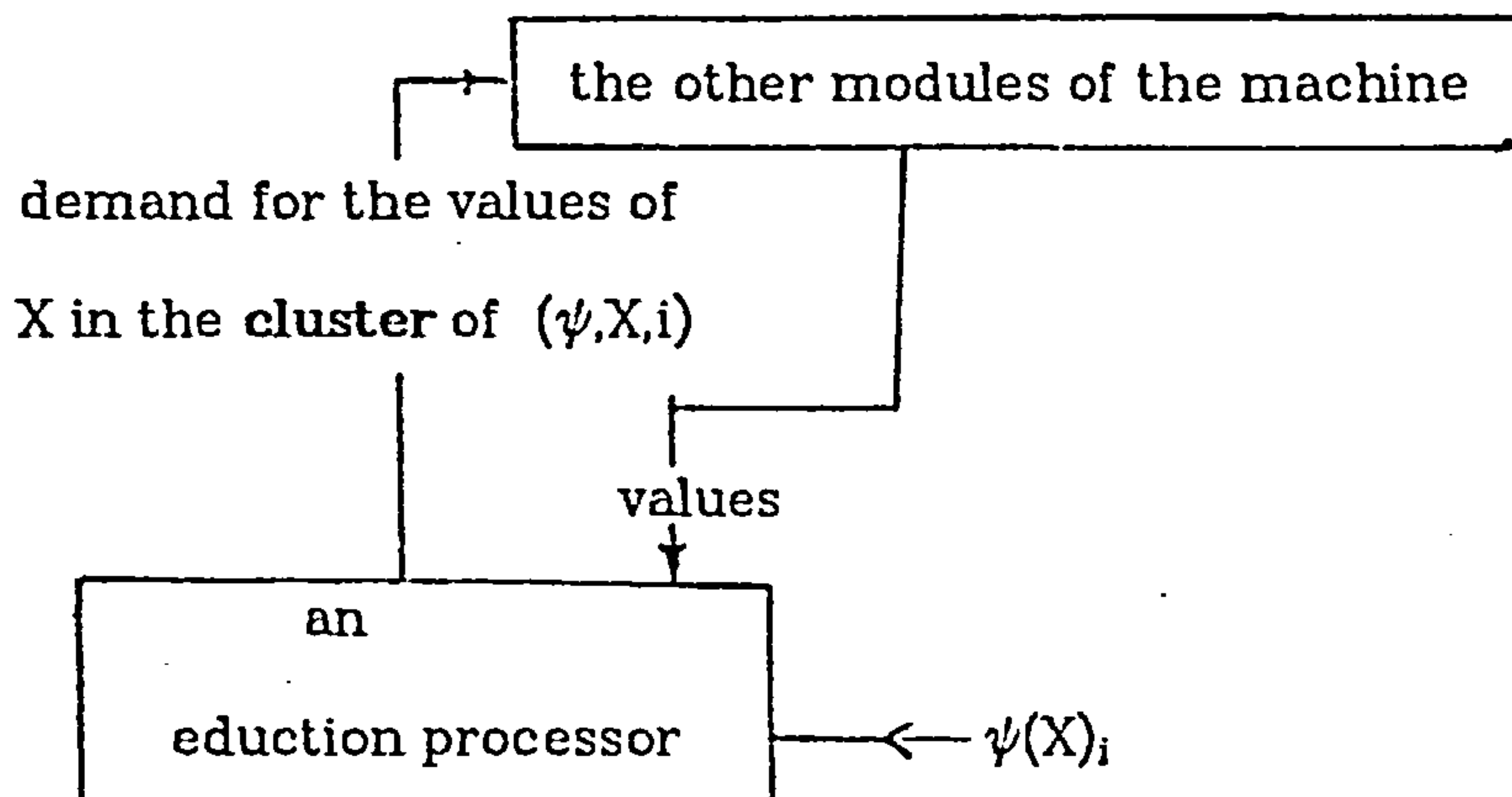
Thus if we want to compute the value of  $X \text{ fby } Y$  at the world 3, for example, we need to know the value of  $X$  at 0, and that of  $Y$  at 2. Therefore, any subset of  $\omega \times \omega$  which contains  $\langle 0, 2 \rangle$  is an element of the galaxy  $\Delta(\text{fby}, \langle X, Y \rangle, 3)$ .

Clearly, the galaxy contains more information about the argument than we need. In the example above, any subset of  $\omega \times \omega$  which contains the element  $\langle 0, 2 \rangle$  is also an element in the galaxy, including the set  $\omega \times \omega$  itself. Thus we may compute values which are not needed after all. Or worse, if the element in the galaxy is taken to be  $\omega \times \omega$ , the machine may sink in an infinite computation. Notice that, for any  $n$ -ary operation, the set  $U^n$  ( $= U \times U \times \dots \times U$   $n$  times) is an element in the galaxy of such an operation, where  $U$  is the universe.

Clearly then, our interest should be with the least element in the galaxy; that is with the least information about the argument. As the elements of the galaxy are partially ordered by the usual set inclusion, the least element is the intersection of all the elements. We shall call such subset the **cluster** of the operation.

Considering the above naive illustration of the eduction machine, the

evaluation process can be illustrated as follows



### Definition

Let  $B(= \langle U, F, D \rangle)$  be an intensional  $\Sigma$ -algebra.

Let  $\psi$  be an  $n$ -ary operation over  $B$ .

Then, for any  $i \in U$  and for any  $X \in {}^U D$ ,

the cluster of  $\psi(X)$  at  $i$ , in symbols  $\Gamma(\psi, X, i)$ , is the intersection of the galaxy of  $\psi(X)$  at  $i$ .

That is  $\Gamma(\psi, X, i) = \bigcap \Delta(\psi, X, i)$

### Examples:

In the algebra  $\mathcal{L}\mathcal{U}(A)$

$$\forall X \forall i \Gamma(\text{first}, X, i) = \{0\}$$

$$\forall X \forall i \Gamma(\text{next}, X, i) = \{j : j = i + 1\}$$

$$\forall X \forall Y \forall i \Gamma(\text{fby}, \langle X, Y \rangle, i) = \{ \langle 0, j \rangle : j = i - 1 \}$$

$$\forall X \forall i \Gamma(\text{hitherto}, X, i) = \{j : j \leq i\}.$$

Of course, we want the cluster of  $\psi(X)$  at  $i$  to determine the least information needed about the argument  $X$  in order to evaluate  $\psi(X)_i$ . We defined the cluster to be the intersection of the elements of the galaxy. This begs the question that the intersection might not be an element in the galaxy; that is the galaxy is not closed under intersection, and hence the information determined



by the cluster is not sufficient to compute the value of  $\psi(X)$  at  $i$ .

We give here an example of an operation whose galaxy at any world in the universe is not closed under intersection, and hence the cluster at any world does not determine the value of the operator at that particular world.

**Example:**

Define the operation PAROR over  $\mathcal{L}\mathcal{U}(N)$  as follows

$$\forall i \in \omega \forall X, Y \in {}^\omega N$$

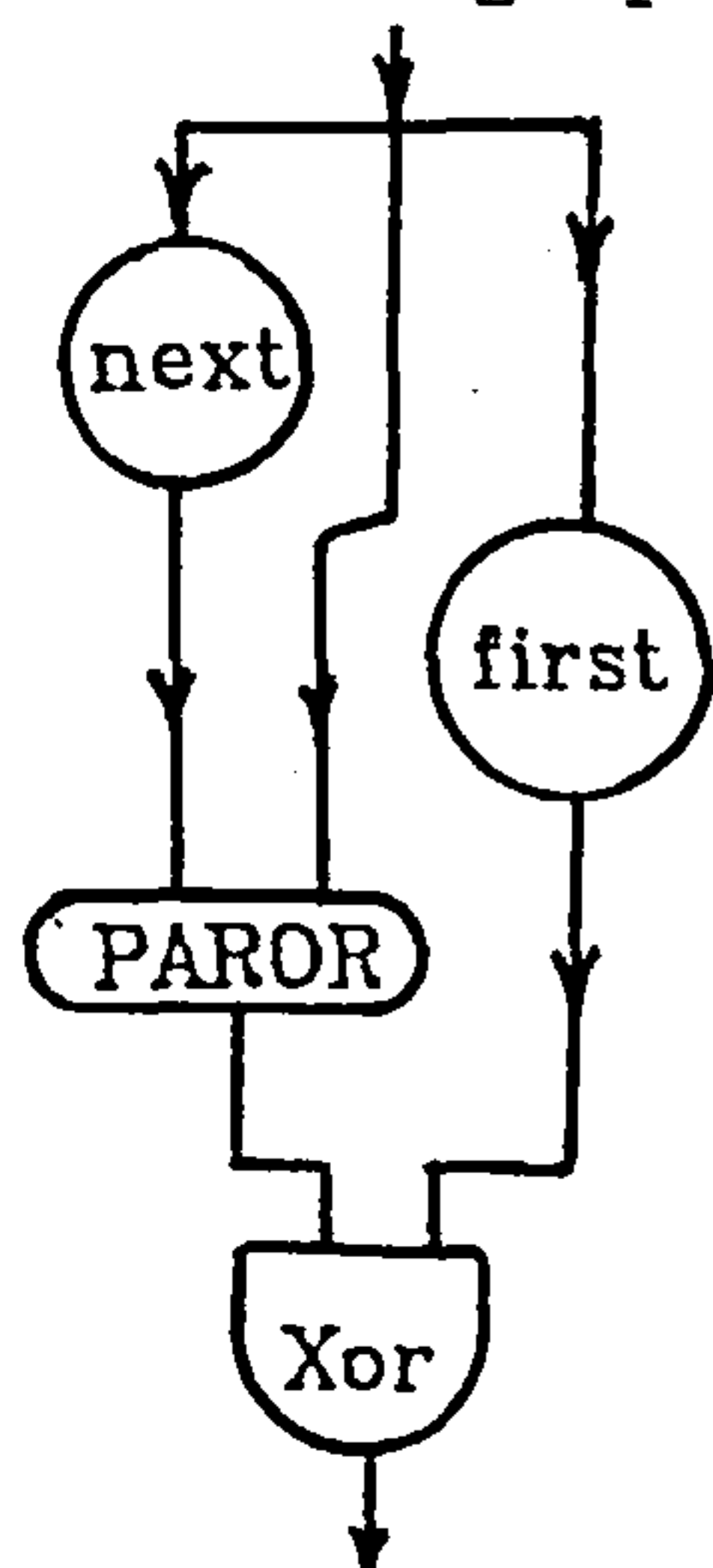
$$\text{PAROR}(X, Y)_i = \begin{cases} \text{true} & \text{if either } X_i \text{ or } Y_i \text{ is true} \\ \text{false} & \text{if } X_i \text{ and } Y_i \text{ are both false} \\ \perp & \text{otherwise} \end{cases}$$

Define the operator PNext over  $\mathcal{L}\mathcal{U}(N)$  as

$$\forall X \in {}^\omega N \forall i \in \omega \quad \text{PNext}(X)_i = \text{PAROR}(\text{next}(X), X)_i \text{ Xor } X_0$$

where Xor is "exclusive or"

That is, the dataflow graph



the dataflow graph for the operator PNext

Clearly, the value of  $\text{PNext}(X)_i$  depends on either  $X_i$  and  $X_0$  or  $X_{i+1}$  and  $X_0$ .

That is,  $\Delta(\text{PNext}, X, i) = \{J \subseteq \omega : 0 \in J \wedge (i \in J \vee i+1 \in J)\}$

Clearly then,  $\Gamma(\text{PNext}, X, i) = \bigcap \Delta(\text{PNext}, X, i) = \{0\} \not\subseteq \Delta(\text{PNext}, X, i)$

Generally speaking, if we allow parallel primitive operators, like PAROR above, then the intersection of the galaxy is not always a member of such galaxy.

In this thesis we shall consider only those operators whose galaxies are closed under intersection. That is, those operators whose clusters are members of their galaxies. We call such operators **functionally sequential operators**.

**Definition:**

Let  $A (= \langle U, F, D \rangle)$  be an intensional  $\Sigma$ -algebra, and let  $\psi$  be an  $n$ -ary operation in  $A$ . Then  $\psi$  is **functionally sequential** iff

$$\forall \langle X_0, \dots, X_{n-1} \rangle \in ({}^U D)^n \forall i \in U$$

$$\Gamma(\psi, \langle X_0, \dots, X_{n-1} \rangle, i) \in \Delta(\psi, \langle X_0, \dots, X_{n-1} \rangle, i)$$

Clearly from the definition, a unary operator  $\psi$  is functionally sequential when

$$\Gamma(\psi, X, i) \in \{J \subset U : \forall Y \in {}^U D \ Y|_J = X|_J \Rightarrow \psi(Y)_i = \psi(X)_i\}$$

It is important to point out that our notion of sequentiality is closed under function composition. That is, if  $\psi$  and  $\varphi$  are functionally sequential, then so are  $\psi \circ \varphi$  and  $\varphi \circ \psi$ . Note also that the cluster might be infinite, this will be discussed in section 2.7.

**Remark:**

According to our definition of a cluster, we can reformulate the definition of synchronic versus asynchronous operators of an intensional algebra in the following definition.

**Definition:**

Let  $A (= \langle U, F, D \rangle)$  be an intensional algebra

An  $n$ -ary operator  $\psi$  over  $A$  is **synchronic** iff

$$\forall i \in U, \forall \langle X_0, \dots, X_{n-1} \rangle \in ({}^U D)^n$$

$$\Gamma(\psi, \langle X_0, \dots, X_{n-1} \rangle, i) = \{ \langle i, i, \dots, i \rangle \}. \quad (\text{i.e. the constant sequence of } i\text{'s}).$$

Otherwise, but not empty, it is asynchronous.

That is,  $\psi$  is synchronic if and only if the value of  $\psi(X_0, \dots, X_{n-1})_i$ , for any  $i$  in the universe  $U$  and any  $n$  values  $\langle X_0, \dots, X_{n-1} \rangle$  in  $(^U D)^n$ , depends on  $i$  and on the values  $X_0(i), \dots, X_{n-1}(i)$ . That is, on  $i$  and on the  $i$ 'th values of its arguments.

Otherwise; that is, if the value of  $\psi(X_0, \dots, X_{n-1})_i$  depends on the  $j$ 'th value of one of its arguments, where  $j \neq i$ , then  $\psi$  is asynchronous.

Notice also that, if  $\Gamma(\psi, \langle X_0, \dots, X_{n-1} \rangle, i)$  is empty, then  $\psi$  is a constant function as it does not depend on the value of any of its arguments.

## 2.7- Computability requirements for intensional algebras

As we are going to use intensional algebras in the semantics of programming languages where arbitrary recursive definitions are allowed, we have to use, only those, algebras which are continuous in the sense of ADJ group [GTW78]. This requirement consists, basically, of two constraints. First, the domain of an intensional algebra should be a CPO. Second, the operations defined by the algebra on its domain should be continuous.

### Proposition

Let  $D$  be a CPO under the partial order  $\sqsubseteq$ , and let  $U$  be a non-empty set. Then  $\prod^U D$  is a CPO under the pointwise extension of  $\sqsubseteq$ .

Proof: Straight forward.

### Proposition

Let  $A (= \langle F, D \rangle)$  be an extensional algebra, and  $U$  be a non-empty set. Then  $A$  is continuous  $\Rightarrow \prod^U A$  is continuous.

Proof: As  $\prod^U D$ , the intensional domain of  $\prod^U A$ , is a CPO under the pointwise extension of  $\sqsubseteq_A$ , then what we need to prove is that all operators in  $\prod^U A$  are continuous. For simplicity we shall denote  $\prod^U A$  by  $P$ .

Without losing generality we prove the proposition for unary operators.

#### 1- Monotonicity:

Let  $X \sqsubseteq_P Y$

then  $\forall i \in U \ X_i \sqsubseteq_A Y_i$

$\Rightarrow \forall i \in U \ \psi_i(X)_i \sqsubseteq_A \psi_i(Y)_i$  ( by the monotonicity of  $\psi_i$  for every  $i$  )

$\Rightarrow \forall i \in U \ \psi(X)_i \sqsubseteq_A \psi(Y)_i$



$$\Rightarrow \psi(X) \subseteq_P \psi(Y)$$

■

2: Let  $\{X_n\}_{n \in \omega}$  be an ascending chain in  ${}^U D$

prove that  $\psi(\sqcup\{X_n\}) = \sqcup\{\psi(X_n)\}$

$$\forall n \in \omega \ X_n \subseteq \sqcup\{X_n\}$$

$$\forall n \in \omega \ \psi(X_n) \subseteq \psi(\sqcup\{X_n\})$$

by the monotonicity of  $\psi$

$$\text{Hence } \sqcup\{\psi(X_n)\} \subseteq \psi(\sqcup\{X_n\})$$

$$\text{Suppose that } \psi(\sqcup\{X_n\}) \not\subseteq \sqcup\{\psi(X_n)\}$$

$$\text{then } \exists i \in U \text{ st } \psi(\sqcup\{X_n\})_i \not\subseteq \sqcup\{\psi(X_n)\}_i$$

$$\Rightarrow \exists i \in U \text{ st } \psi_i(\sqcup\{X_n\})_i \not\subseteq \sqcup\{\psi_i(X_n)\}_i$$

$$\text{then } \psi_i(\sqcup\{X_n(i)\}) \not\subseteq \sqcup\{\psi_i(X_n(i))\}$$

then  $\psi_i$  is not continuous. (contradiction)

$$\text{Therefore } \psi(\sqcup\{X_n\}) \subseteq \sqcup\{\psi(X_n)\}.$$

and hence  $\psi$  is continuous.

■

The proof of the continuity of the pointwise operators was straight forward as the problem is reducible to the continuity of their extensional meaning. This is due to the fact that the value of a pointwise operator at any index in the universe depends only on the extensional values of its arguments at that same index.

The case, however, is different for non-pointwise operators as the value of an operator in a certain world (at a certain index) depends on the index itself together with the intensional values of its arguments. In particular, the value of an expression  $\psi(x)$  in a world  $i$  depends on the value of  $x$  in the cluster  $\Gamma(\psi, x, i)$ . Notice that, we are considering functionally sequential

operators. That is, operators for which the cluster is a member of the galaxy. More specifically, the cluster determines the least essential information (about the argument) needed to compute the value of the operator in that particular world.

In evaluating by eduction, the most essential point we have to stress is the finiteness property; if  $\Gamma(\psi, x, i)$  is infinite then  $\psi(x)_i$  is not computable. This is formalized in the theorem below.

Informally speaking, in computing  $\psi(x)_i$  we need only a finite amount of information about  $x$ ; i.e. we need to compute the value of  $x$  in only a finite subset of the universe. Clearly, otherwise means that the computation of  $\psi(x)_i$  will live-lock waiting for the infinite computation of the argument  $x$ .

Another point to emphasize here is that the extensional domains of the algebras we are working with should be flat. That is, we deal with intensional algebras of the form  $\langle U, F, D \rangle$  where  $D$  is flat; i.e.

$$\forall x, y \in D \quad x \sqsubseteq y \Leftrightarrow x = \perp \text{ or } x = y.$$

The partial order on  ${}^U D$  is the pointwise extension of the above order. Informally speaking, this means that the objects of our domains are all complete (there are no partial objects). Thus, the value of an expression in a certain world is either fully computed or not at all. The following proposition is a direct consequence of such an assumption.

**Proposition:**

Let  $\langle U, F, D \rangle$  be an intensional algebra, where  $D$  is a flat domain.

Let  $\{X_n\}_{n \in \omega}$  be an ascending chain in  ${}^U D$

Let  $X_\omega = \sqcup \{X_n\}_{n \in \omega}$ .

Then  $\forall i \in U \exists j \in \omega$  such that  $X_\omega(i) = X_j(i)$

**Proof:** Straight forward from the assumption that  $D$  is a flat domain.

**Lemma 2.1:**

Let  $A (= \langle F, D \rangle)$  be an extensional  $\Sigma$ -algebra, and  $U$  be a non-empty set.

Let  $C$  be an intensional  $\Sigma$ -algebra such that  $A^U \subset C$

( $A^U$  is a restriction of  $C$ ).

Let  $\psi$  be an  $n$ -ary monotonic operation over  $C$ . Then

$\psi$  continuous  $\Rightarrow \forall \langle x_0, \dots, x_{n-1} \rangle \in ({}^U D)^n \forall i \in U$

$(\psi(\langle x_0, \dots, x_{n-1} \rangle)_i \neq \perp \Rightarrow \Gamma(\psi, \langle x_0, \dots, x_{n-1} \rangle, i) \text{ is finite})$

**Proof:**

We prove the case here for unary operators and for  $U = \omega$ . That is,

$\psi$  continuous  $\Rightarrow \forall X \in {}^\omega D \forall i \in \omega (\psi(X)_i \neq \perp \Rightarrow \Gamma(\psi, X, i) \text{ is finite})$

Let  $\psi$  be a continuous unary function, and suppose that

$\exists X \in {}^\omega D \exists i \in \omega \psi(X)_i \neq \perp$  and  $\Gamma(\psi, X, i)$  is not finite.

Then,  $\forall J \text{ finite} \subset \omega \ J \neq \Gamma(\psi, X, i)$ ,

that is,  $\forall J \text{ finite} \subset \omega \exists X'_J \in {}^\omega D$  st  $X'_J|_J = X|_J$  and  $\psi(X'_J)_i \neq \psi(X)_i$

(i.e.  $\forall J \text{ finite} \subset \omega \ J$  cannot be the cluster)

Define the following  $\omega$ -chain of elements of  ${}^\omega D$

$$X'_0 \sqsubseteq X'_1 \sqsubseteq X'_2 \sqsubseteq \dots \sqsubseteq X'_n \sqsubseteq \dots \sqsubseteq X'$$

such that  $\forall n \in \omega$  define  $X'_n$  as follows

$$\forall j \in \omega, (X'_n)_j = \begin{cases} X_j & \text{if } j < n \\ \perp & \text{if } j \geq n \end{cases}$$

That is, construct the  $\omega$ -chain :

$$\vdots$$

$$X'_n = \langle X_0, X_1, X_2, X_3, \dots, X_n, \perp, \perp, \perp, \dots \rangle$$

$\vdots$

$$X'_4 = \langle X_0, X_1, X_2, X_3, \perp, \perp, \perp, \perp \rangle$$

$$X'_3 = \langle X_0, X_1, X_2, \perp, \perp, \perp, \perp \rangle$$

$$X'_2 = \langle X_0, X_1, X_n, \perp, \perp, \perp, \dots \rangle$$

$$X'_1 = \langle X_0, \perp, \perp, \perp, \perp, \perp, \dots \rangle$$

$$X'_0 = \langle \perp, \perp, \perp, \perp, \perp, \perp, \dots \rangle$$

From the definition of the  $\omega$ -chain  $X'_n$ ,  $\sqcup X'_n = X$

therefore by the monotonicity of  $\psi$ ,  $\psi(\sqcup X'_n) = \psi(X)$  .....(\*1)

Now,  $\forall n \in \omega$   $X'_n|_n = X|_n$  (notice here that  $n = \{k: k < n\}$ )

and because  $n$  is finite, then  $n \neq \Gamma(\psi, X, i)$

therefore  $\forall n \in \omega$   $\psi(X'_n)_i \neq \psi(X)_i$

Since  $D$  is a flat domain, then from the proposition above we have

$$\sqcup \psi(X'_n)_i \neq \psi(X)_i$$

Therefore  $\sqcup \psi(X'_n) \neq \psi(X)$  ....(\*2)

From (\*1) and (\*2) then

$$\psi(\sqcup X'_n) = \psi(X) \neq \sqcup \psi(X'_n)$$

that is,  $\psi(\sqcup X'_n) \neq \sqcup \psi(X'_n)$

Therefore  $\psi$  is not continuous, which contradicts our supposition. ■

### Lemma 2.2:

Let  $A$  ( $= \langle F, D \rangle$ ) be an extensional  $\Sigma$ -algebra, and  $U$  be a non-empty set.

Let  $C$  be an intensional  $\Sigma$ -algebra such that  $A^U \subset C$

( $A^U$  is a restriction of  $C$ ).

Let  $\psi$  be an  $n$ -ary monotonic operation over  $C$ . Then

$$\psi \text{ continuous} \Leftarrow \forall \langle x_0, \dots, x_{n-1} \rangle \in ({}^U D)^n \quad \forall i \in U$$

$$(\psi(\langle x_0, \dots, x_{n-1} \rangle))_i \neq \perp \Rightarrow \Gamma(\psi, \langle x_0, \dots, x_{n-1} \rangle, i) \text{ is finite})$$

### Proof:

Let  $\psi$  be a monotonic function, and assume that

$$\forall X \in {}^\omega D \text{ and } \forall i \in \omega \quad (\psi(X))_i \neq \perp \Rightarrow \Gamma(\psi, X, i) \text{ is finite. ...}(*1)$$

Prove that  $\psi$  is continuous.

Let  $\langle X_i \rangle_{i \in \omega}$  be an ascending chain over the elements of  ${}^\omega D$ , and let  $\sqcup X$  be the least upper bound of the chain.

$$\text{Then, } \forall i \in \omega \quad X_i \sqsubseteq \sqcup X$$

then by the monotonicity of  $\psi$ ,  $\forall i \in \omega \quad \psi(X_i) \sqsubseteq \psi(\sqcup X)$

therefore  $\sqcup \psi(X_i) \sqsubseteq \psi(\sqcup X)$



Suppose that  $\sqcup \psi(X_i) \sqsubset \psi(\sqcup X)$  (i.e.  $\sqcup \psi(X_i) \neq \psi(\sqcup X)$ )

then  $\exists j \in \omega$  such that  $(\sqcup \psi(X_i))_j \sqsubset (\psi(\sqcup X))_j$

therefore  $\forall i \in \omega \ \psi(X_i)_j \sqsubset \psi(\sqcup X)_j$

Since  $D$  is a flat domain, then

$\forall i \in \omega \ \psi(X_i)_j \sqsubset \psi(\sqcup X)_j \Rightarrow \forall i \in \omega \ \psi(X_i)_j = \perp$  and  $\psi(\sqcup X)_j \neq \perp \dots (*2)$

By the assumption (\*1) above,  $\psi(\sqcup X)_j \neq \perp \Rightarrow \Gamma(\psi, \sqcup X, j)$  is finite.

Let  $\Gamma(\psi, \sqcup X, j) = J$ , and let  $Y \in {}^\omega D$  such that  $Y|_J = \sqcup X|_J$

Define  $Y'$  as follows,  $\forall i \in \omega \ Y'_i = \begin{cases} Y_i & \text{if } i \in J \\ \perp & \text{otherwise} \end{cases}$

Since  $J$  is finite, then  $\exists k \in \omega$  such that  $Y' \sqsubseteq X_k$

By the monotonicity of  $\psi$ ,  $Y' \sqsubseteq X_k \Rightarrow \psi(Y') \sqsubseteq \psi(X_k)$

By (\*2) above then,  $\psi(Y')_j \sqsubseteq \psi(X_k)_j = \perp$

By the definition of  $J$ ,  $\psi(Y')_j = \psi(Y)_j = \psi(\sqcup X)_j$

Hence,  $\psi(\sqcup X)_j = \psi(Y)_j \sqsubseteq \psi(X_k)_j = \perp$ . This contradicts (\*2) above.

Therefore  $\sqcup \psi(X_i) = \psi(\sqcup X)$ . i.e.  $\psi$  is continuous. ■

### Theorem: *The finiteness Theorem*

Let  $A$  ( $= \langle F, D \rangle$ ) be an extensional  $\Sigma$ -algebra, and  $U$  be a non-empty set.

Let  $C$  be an intensional  $\Sigma$ -algebra such that  $A^U \subset C$

( $A^U$  is a restriction of  $C$ ).

Let  $\psi$  be an  $n$ -ary monotonic operation over  $C$ . Then

$\psi$  continuous  $\Leftrightarrow \forall \langle x_0, \dots, x_{n-1} \rangle \in ({}^U D)^n \ \forall i \in U$

$(\psi(\langle x_0, \dots, x_{n-1} \rangle))_i \neq \perp \Rightarrow \Gamma(\psi, \langle x_0, \dots, x_{n-1} \rangle, i)$  is finite)

**Proof:** By Lemma 2.1 and Lemma 2.2. ■

## 2.8- The Target Language *DE*

We introduce here the family of intensional programming languages *DE*, for **Definitional Equations**. It is a family in the sense that it is, like *Iswim* and *Lucid*, a language constructor which maps algebras to languages. It is the family

$$DE = \{ DE(A) \mid A \text{ is an intensional algebra} \}.$$

Informally speaking, a program written in a member of *DE* is a set of compatible equations defining nullary variables. Each equation is of the form

$$\text{variable} = \text{expression}$$

Since a program is a set, the order in which the definitions are written is insignificant. The set is compatible means that each variable is defined at most once. One of these equations must define the variable **result**. So, neither function definitions nor Algol-like block structures is allowed in *DE*. We say that members of the family *DE* are intensional languages because the construct *DE* expects intensional algebras as arguments. Moreover, members of *DE* are typeless, and a variable symbol, in a member of *DE*, can be assigned any object in the domain of that instance without the need of type declaration.

Members of *DE* will be used as target languages into which we translate the functional language to be interpreted. Our main interest is with functional programming languages in general, and the families *Lucid* and *Iswim* in particular. Informally speaking, the idea of compiling the source language to a member in *DE* is to resolve the complexities of the source language, like block structures, function definitions, and so on. These complexities will be taken care of by intensional operators in the algebra on which *DE* is to be applied. So, the target language will be a simple equational definitional language on intensional algebras. These intensional algebras must be based on the pointwise intension of the algebra of the source language. That is, it extends the operators of this algebra pointwise, then enriches the pointwise intensional algebra by some

intensional operators which will resolve the above mentioned complexities.

We introduce here the abstract syntax for  $DE$ . As  $DE$  is a language constructor (a family) then both the syntax and the semantics of each member  $DE(A)$  is determined by the intensional algebra  $A$ .

### 2.8.i- The syntax of $DE$

Let  $A$  be an intensional algebra. The abstract syntax of  $DE(A)$  is defined as follows:

$DE(A)$ -expressions:

The set of  $DE(A)$ -expressions is the smallest set  $X$  such that

- All nullary variable symbols are in  $X$ .
- If  $\psi$  is an  $n$ -ary constant symbol in the signature of  $A$  and  $\xi_0, \dots, \xi_{n-1}$  are in  $X$  then  $\psi(\xi_0, \dots, \xi_{n-1})$  is in  $X$

$DE(A)$ -equations

A  $DE(A)$ -equation consists of a left hand side (lhs) which is a nullary variable symbol, and a right hand side (rhs) which is a  $DE(A)$ -expression. i.e. a  $DE(A)$ -equation is of the form

$\mathbf{v} = DE(A)\text{-expression}$  where  $\mathbf{v}$  is a nullary variable symbol.

$DE(A)$ -programs

A  $DE(A)$ -program is a set of  $DE(A)$ -equations. such that

- \*- Compatibility: a variable symbol is bound at most once in the whole set; i.e. it does not occur as a lhs of more than one equation.
- \*- One of these equations defines the variable symbol **result**,

### Occurrences of Variables in $DE$ :

Clearly, occurrences of variable symbols in  $DE(A)$ -expressions are all free. In a  $DE(A)$ -program however, an occurrence of a variable symbol is bound if that symbol occurs as a lhs of an equation in the program; otherwise it is a free occurrence.

### 2.8.ii- The Semantics of $DE$

Besides determining the syntax, the intensional algebra  $A$  determines also the semantics of  $DE(A)$ . In addition to  $A$  however, we need an intensional environment to assign meaning to the variable symbols. Thus, the meaning of an expression in  $DE(A)$  is determined by the algebra  $A$  together with an  $A$ -intensional environment. As  $DE$  is not a functional language then such an environment is going to be 'simple' in the sense that it assigns meaning to nullary variable symbols only.

#### $DE(A)$ -Expressions:

Given an intensional algebra  $A$ , and an  $A$ -intensional environment  $\varepsilon$ ; the semantics of a  $DE(A)$ -expression  $\tau$ , denoted by  $\models_{A,\varepsilon} \tau$  is defined recursively as follows:

- a: If  $\tau$  is a nullary variable symbol, then the value of  $\tau$  in the environment  $\varepsilon$  is  $\varepsilon(\tau)$ ; i.e. the value assigned to  $\tau$  by the environment  $\varepsilon$ .

$$(\models_{A,\varepsilon} \tau) = \varepsilon(\tau)$$

- b: If  $\tau$  is of the form  $\psi(x_0, \dots, x_{n-1})$  where  $\psi$  is an  $n$ -ary constant symbol, and  $x_0, \dots, x_{n-1}$  are  $DE(A)$ -expressions, then the value of  $\tau$  in the environment  $\varepsilon$  is the value of  $\psi$  assigned to it by the algebra  $A$  applied to the values of  $x_0, \dots, x_{n-1}$  relative to the algebra  $A$  and the environment  $\varepsilon$ .

$$\text{i.e. } (\models_{A,\varepsilon} \tau) = A(\psi) (\models_{A,\varepsilon} x_0, \dots, \models_{A,\varepsilon} x_{n-1})$$



*Definition:*

Given an  $A$ -intensional environment  $\varepsilon$ , and a  $DE(A)$ -equation  $v = \text{Exp}$ , We say that  $\varepsilon$  satisfies the equation if and only if

$$\varepsilon(v) = \models_{A,\varepsilon} \text{Exp}$$

We say that  $\varepsilon$  satisfies the  $DE(A)$ -program  $P$  if and only if  $\varepsilon$  satisfies all the equations of  $P$  simultaneously.

*DE(A)-Programs:*

Given an intensional algebra  $A$ , the value of a  $DE(A)$ -program  $P$  in the  $A$ -intensional environment  $\varepsilon$ , is the value of the variable **result** in the least environment  $\varepsilon'$ , such that  $\varepsilon'$  satisfies the equations of  $P$  and agrees with  $\varepsilon$  except at most on the values assigned to the local variable symbols of  $P$ .

It is worth pointing out here that if the extensional domain of  $A$  is a CPO, then so is the intensional domain under the pointwise extension of the partial order defined on the extensional domain. This induces a partial order on the set of all  $A$ -intensional environments. That is,

if  $\varepsilon, \varepsilon' \in [V \rightarrow {}^U D]$ , where  $D$  and  $U$  are, resp., the

extensional domain and the universe of the algebra  $A$ ;

and  $V$  is the set of variable symbols, then

$$\varepsilon < \varepsilon' \Leftrightarrow \forall v \in V \forall u \in U (\varepsilon(v)_u <_D \varepsilon'(v)_u)$$

Hence, if  $D$  is a CPO then the least environment does exist.

*Example:*

From the definition of  $DE$  and the intensional algebra  $L\mathcal{U}$ , it is clear that the language **BLucid** is a subfamily of  $DE$ . It is the subfamily  $DE(L\mathcal{U})$ . Let  $Z$  be the usual mathematical algebra over the integers. Then the following is a

program in  $DE(Lu(Z))$  or equivalently in  $BLucid(Z)$

**result = I-1 asa (J > first n);**

**I = 0 fby I + 1 ;**

**J = 0 fby J + 2\*I + 1 ;**

Clearly, given a  $Z$ -intensional environment  $\varepsilon$ , the value of the above program in  $\varepsilon$  is the integer square root of the value  $\varepsilon(n)_0$  if such a value is positive. If  $\varepsilon(n)_0$  is negative however, the value of the program is  $-1$ .

## Chapter 3

# Compiling Functions into Intensional Logic

### 3.0 Introduction

In this chapter we describe the procedure of compiling non-structured functional languages into our intensional target language *DE*. We give, also, the basic concepts in our technique. Such a procedure will be described in two main steps.

Firstly (sections 3.1-3.6), we describe the compilation of function definitions and calls into the target language *DE*. The source language we are going to consider here is a very simple subfamily of *Iswim*. It is the family *Iwade* = {*Iwade*(A) : A is an algebra}.

*Iwade* is the subfamily of *Iswim* in which:

(1) function definitions do not have global occurrences of nullary variable symbols. Thus for example, the following is not a legitimate function definition in *Iwade*

$$F(X) = X + A$$

because *A* occurs as a global in the definition of *F*.

(2) The where-expression does not appear as a right hand side of an equation. So, for example, the following is not a legitimate definition in *Iwade*

$$G(X,Y) = Z \text{ where}$$

$$Z = X * X + 2 * Y;$$

end

For an extensional algebra *A*, the language *Iwade*(*A*) will be compiled into the target *DE*(*FUN*(*A*)). The algebra *FUN*(*A*), which we shall define later, is the

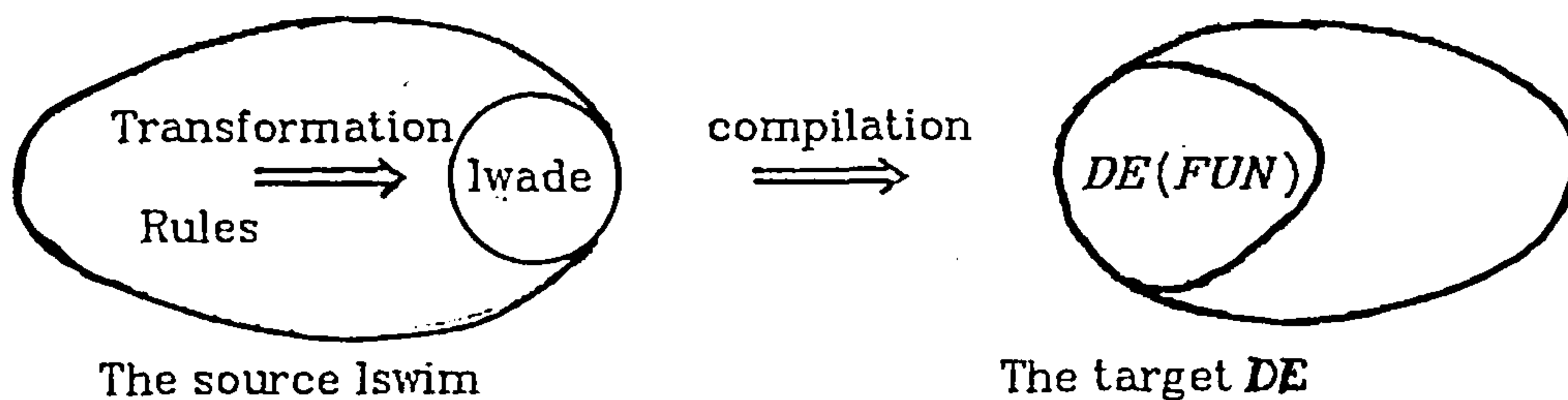
intensional algebra based on the point wise extension of  $A$  upon the universe of function calls and will contain intensional operators which resolve the complexities associated with function calls.

We should emphasize here that the technique described is enough, on its own, to implement a structured functional language, like *Iswim*. We can massage *Iswim*-programs to programs in *Iwade* using a collection of program transformation rules [AsWa79] [WaAs80]. These are purely syntactic manipulation rules for programs in *Iswim*. The transformation according to these rules is a logical derivation; i.e a sequence of equivalent programs  $P_0, \dots, P_{n-1}$  where  $P_0$  is the original program and for any  $i \in n$

$P_{i+1}$  is obtained from  $P_i$  by one of the manipulation rules.

We shall sketch these rules in section 3.2.

Thus the procedure of compiling *Iswim* into  $DE(FUN)$  can be illustrated in the following diagram.



**Secondly** (section 3.7), we extend our technique by considering a bigger subset of *Iswim* than *Iwade*. We define the language **Ipaddle** to be the subset of *Iswim* which contains *Iwade* and allows global occurrences of variable symbols in function definitions. That is, the only difference between **Ipaddle** and *Iswim* is that the latter is a structured language, while no nesting is allowed in the former. We extend the algebra  $FUN$ , and compile **Ipaddle** to the target  $DE(FUN)$  where  $FUN$  here is such an extension.



### 3.1 The Abstract Syntax of Iwade:

Given a  $\Sigma$ -extensional algebra  $A$ , a program in  $Iwade(A)$  is either a **simple expression** or a **where-expression**.

The set of **simple expressions** in  $Iwade(A)$  is defined recursively as follows

If  $\alpha$  is either an  $n$ -ary constant symbol in  $\Sigma$

or an  $n$ -ary variable symbol, and  $x_0, \dots, x_{n-1}$  are

simple expressions, then  $\alpha(x_0, \dots, x_{n-1})$

is a simple expression.

If  $\varepsilon$  is a simple expression in  $Iwade(A)$  and  $\delta_0, \dots, \delta_{m-1}$  are  $m$  compatible valid definitions, then

$\varepsilon$  **where**

$\delta_0$

$\dots$

$\delta_{m-1}$

**end**

is a **where-expression**.

A **valid definition** in  $Iwade$  is of the form

$$\vartheta(\eta_0, \dots, \eta_{n-1}) = \varepsilon$$

where  $\vartheta$  is an  $n$ -ary variable symbol,

$\eta_0, \dots, \eta_{n-1}$  are  $n$  distinct and different from  $\vartheta$

nullary variable symbols (called the **formals** of  $\vartheta$ ),

and  $\varepsilon$  is a simple expression in which all nullary

variable symbols occur are **formals** in the definition.

A set of valid definition is **compatible** if each variable symbol is bound at most once in the whole set; i.e. occurs as a left hand side or as a formal in a function

definition only once.

Clearly, Iwade is a subfamily of the language Iswim. For any algebra  $A$ , a program in  $Iwade(A)$  is also a program in  $Iswim(A)$ . Moreover, we define the semantics of Iwade to be that of Iswim, i.e. the meaning of a construct in  $Iwade(A)$ , for an algebra  $A$ , is the same as its meaning according to the semantics of  $Iswim(A)$ .

### 3.2 From Iswim to Iwade

Our main interest in this chapter is to compile the subfamily Iwade into the target DE. However, we sketch briefly in this section the program transformation rules which we can employ to transform Iswim-programs into programs in Iwade. These rules are described formally in [AsWa79].

#### 3.2.1- The Renaming Rule:

This rule states, informally, that local variables in a clause are all dummy variables and can be renamed as far as the renaming process is safe. That is, if it does not change a free occurrence of a variable into a bound occurrence, and it preserves the compatibility of the resulted set of definitions of the clause.

A *renaming* is a non-ambiguous set of pairs of variable symbols.

*Non-ambiguous* means that no variable symbol occurs as the left hand side of more than one renaming pair. We shall represent a pair  $\langle x, y \rangle$  in a renaming by  $x \leftarrow y$ . Simply,  $x$  is the present variable symbol and  $y$  is the new symbol. The set, for example,

$$\{ v \leftarrow w, a \leftarrow b, v \leftarrow z \}$$

is not a renaming as  $v$  is renamed once as  $w$  and once as  $z$ .

If  $\tau$  is an Iswim expression, and  $E$  is a renaming, then we write  $R(E, \tau)$  for the result of replacing the variables of  $\tau$  which occur in the left hand side of  $E$  by their corresponding right hand sides.

Example:

If  $\tau$  is  $A + B - C$

and  $E$  is  $\{v \leftarrow w, A \leftarrow q, C \leftarrow P\}$

then  $R(E, \tau)$  is  $q + B - P$

For an Iswim-expression  $\tau$ , a renaming  $E$  is said to be *safe* on  $\tau$  when no free occurrences in  $\tau$  becomes bound in  $R(E, \tau)$ . For a where-expression in Iswim, we need another condition to insure the safety of the renaming process. We do not want to introduce a definition for a variable symbol which is already defined in the clause. For example, the renaming  $\{b \leftarrow x\}$  on the expression

$x+y$  where

$x = y + b;$

$b = 10;$

end

is not safe, as it results with two definitions for  $x$ . Thus a stronger condition for the safety of a renaming  $E$  on an expression  $\tau$  is, that no variable in the right hand side of  $E$  should occur either bound or free in  $\tau$ .

A variant of this rule is the *formal parameter renaming rule*. This states that the formal parameters in a function definition are dummies and can be renamed provided that the renaming process does not change a free occurrence of a variable to a bound occurrence. For example the renaming  $\{x \leftarrow a\}$  is not safe on the definition

$F(x) = a + x - b$

as it changes the free occurrence of  $a$  into a bound occurrence (a formal of  $F$ ).

### 3.2.2- The Amalgamation Rule:

This allows us to amalgamate an expression consisting of several where-clauses into one single clause. For example, the Iswim(Z) expression

$$\left[ \begin{array}{l} \mathbf{x*y \ where} \\ \mathbf{x=10;} \\ \mathbf{y=2;} \\ \mathbf{end} \end{array} \right] + \left[ \begin{array}{l} \mathbf{z+r \ where} \\ \mathbf{z=5;} \\ \mathbf{r=9;} \\ \mathbf{end} \end{array} \right]$$

can be amalgamated into

$(\mathbf{x*y}) + (\mathbf{z+r}) \ \mathbf{where}$

$\mathbf{x=10;}$

$\mathbf{y=2;}$

$\mathbf{z=5;}$

$\mathbf{r=9;}$

$\mathbf{end}$

The rule states that if  $\Theta(C_0, \dots, C_{n-1})$  is an expression, and  $C_0, \dots, C_{n-1}$  are  $n$  where-clauses such that the union of the definitions of these clauses is compatible. Then the clauses can be amalgamated into one single clause  $B$  where the subject of  $B$  is  $\Theta(c_0, \dots, c_{n-1})$  and  $c_i$  is the subject of  $C_i$  for every  $i$ . The body of  $B$  is the union of the definitions of the bodies of all clauses.

Notice that if the union is not compatible then we apply the renaming rule before amalgamation. For example, the expression

$$\left[ \begin{array}{l} \mathbf{a+b \ where} \\ \mathbf{a=x+y;} \\ \mathbf{b=2;} \\ \mathbf{x=10;} \\ \mathbf{end} \end{array} \right] \times \left[ \begin{array}{l} \mathbf{a+x \ where} \\ \mathbf{a=y+3;} \\ \mathbf{y=4;} \\ \mathbf{end} \end{array} \right] + \left[ \begin{array}{l} \mathbf{x \ where} \\ \mathbf{x=10;} \\ \mathbf{end} \end{array} \right]$$



cannot be amalgamated without applying the renaming rule to make the sets of definitions in the bodies of the clauses compatible. By applying the renaming

$\{a \leftarrow z, y \leftarrow v, x \leftarrow m\}$  on the second clause, and

$\{x \leftarrow n\}$  on the third, we get

$$\left[ \begin{array}{l} a+b \text{ where} \\ \quad a=x+y; \\ \quad b=2; \\ \quad x=10; \\ \text{end} \end{array} \right] * \left[ \begin{array}{l} z+m \text{ where} \\ \quad z=v+3; \\ \quad v=4; \\ \text{end} \end{array} \right] + \left[ \begin{array}{l} n \text{ where} \\ \quad n=10; \\ \text{end} \end{array} \right]$$

then we amalgamate the clause into

$(a+b)*(z+m)+n$  where

$a=x+y;$

$b=2;$

$x=10;$

$z=v+3;$

$v=4;$

$n=10;$

end

### 3.2.3- The Liquidation Rule

This is similar to the amalgamation rule. It allows us to liquidate (or to amalgamate) an inner where-clause into the first outer one as far as this liquidation is safe. That is, as far as the body of the result clause is a set of compatible equation, and no free variable becomes bound as a result of the amalgamation process. If the process is not safe, then we apply the renaming rule before amalgamation.

Formally, given a term  $\tau$  of the form

$\delta$  where  $D$  end

where  $D$  is a set of definitions with a definition of the form

$x = \delta'$  where  $D'$  end

then the liquidation of  $\tau$  is the term

$\delta$  where  $D''$  end

where  $D''$  is the union of  $D$  and  $D'$  in which

$x = \delta'$

*Example :* consider the program

$f(y)+y$  where

$y = 10+a;$

$x = 5;$

$f(x) = v + y$  where

$v = a*x;$

$a = x+y;$

$y = 10;$

end;

end

Any attempt to liquidate the inner where-clause into the outer one will give us the following three side effects:

(1) As  $a$  occurs free in the outer where-clause, and local in the inner one, the liquidation makes the free occurrence of  $a$  in  $y=10+a$  bound.

(2) As  $y$  is defined in the two clauses, the liquidation will yield two definitions for the variable symbol  $y$ .

(3)  $x$  occurs local in the main where-clause, and a formal in the definition of  $f$ . The liquidation will result in two definition of  $x$ ; the formal of  $f$  and the local in the main clause. That is, the resulted set of definitions will not be compatible.

Any of these conditions is enough to make us postpone liquidation, as it is not safe, till we do some renaming. We apply the renaming rules to insure the safety of the liquidation process. Using the formal parameter renaming rule, let us apply the renaming  $\{x \leftarrow w\}$  on the formals of  $f$ . The definition of  $f$  then becomes

$f(w) = v + y$  where

$v = a * w;$

$a = w + y;$

$y = 10;$

end;

To solve the problem of the first two side effects, we apply (for example) the renaming

$\{a \leftarrow b, y \leftarrow z\}$

on the inner clause. We said 'for example' because any renaming which does not convert a free occurrence to a bound one is safe and can be applied.

After applying the above renaming on the inner clause, we can liquidate the

inner where-clause into the main one as follows

<pre> f(y)+y where   y = 10+a ;   x = 5 ;   f(w) = v+z where     v = b*w ;     b = w+z ;     z = 10 ;   end ; end </pre>	liquidation =====>	<pre> f(y)+y where   y = 10+a ;   x = 5 ;   f(w) = v+z ;   v = b*w ;   b = w+z ;   z = 10 ; end </pre>
--	-----------------------	--

### 3.2.4- The Calling Rule:

This rule expresses the fact that definitions in the lswim family are transparent; i.e the right hand side of a definition equals the left hand side. It says that if a where-clause contain the definition

$$f(x_0, \dots, x_{n-1}) = \varepsilon$$

where  $f$  is an  $n$ -ary variable symbol,  $x_0, \dots, x_{n-1}$  are the formals of the definition, and  $\varepsilon$  is an expression; then any occurrence of  $f(x_0, \dots, x_{n-1})$  in the clause can be substituted by the expression  $\varepsilon$  as far as such a substitution does not change a free variable in  $\varepsilon$  into a bound variable.

For example, the expression

```

F(3,Y) + F(5,C) where
  C = 2*Y ;
  F(X,Y) = X*X - Y;
end

```



can be transformed into

$(3*3 - Y) + (5*5 - (2*Y))$  where

$C = 2*Y ;$

$F(X,Y) = X*X - Y;$

end

### 3.2.5- The Addition and Deletion Rules:

The **Addition rule** allows us to add to the body of a clause any definition as far as the new defined variable is not already a local in the clause (because this makes the set of definitions incompatible) and as far as it does not make a free variable bound in the new set. For example, in the above expression we can add any definition to the body as far as the left hand side of the definition is not  $C$  or  $F$ .

The **deletion rule** is the opposite of the addition rule. It allows us to discard any definition in the clause provided that such a deletion does not make a bound variable free. By deletion, for example, the above expression

$(3*3 - Y) + (5*5 - (2*Y))$  where

$C = 2*Y ;$

$F(X,Y) = X*X - Y;$

end

becomes  $(3*3 - Y) + (5*5 - (2*Y))$

### 3.2.6- The Formal Parameter Approach to Globals:

The program transformation rules described above can be used only in the linearization of Iswim-programs; i.e. to get rid of the modular structure of Iswim-programs. As function definitions in Iwade do not have global occurrences of nullary variable symbols, then in order to transform Iswim-programs to programs in Iwade we need to get rid of nullary globals in function definitions.

In the formal parameter approach, we remove global occurrences by converting them into formal parameters. Consider, for example, the following set of definitions

$$G(B) = A + B ;$$

$$A = 15 ;$$

The value of  $G$  in any environment is a function of the value which its formal takes in such an environment; and also a function of the value of the global  $A$  in that same environment. So, instead of the above definition of  $G$  we can write

$$G(B,A) = A + B ;$$

and convert  $G$  from a unary function into a binary one, where the second formal is the global to its definition. Generally speaking, an  $n$ -ary function whose definition contain  $m$  global nullary variable symbols is converted to a function of degree  $m+n$ .

Any call to  $G$  above, then, should be a call to a binary function, where the second actual of the call is, what was the global to the definition.

We should draw the attention here to the fact that any deglobalization step (converting a global into a formal) will lead to incompatibility. For instance, deglobalizing  $A$  in the definition of  $G$  will yield two definitions for  $A$ ; a formal to  $G$  and a local in the set of definitions.

$$G(B,A) = A + B ;$$

$$A = 15 ;$$

The way out, is to apply the formal parameter renaming rule and rename  $A$  in the definition of  $G$ . Renaming it as  $C$ , for example, will yield

$$G(B,C) = C + B;$$

and hence, the equations become

$$G(B,C) = C + B;$$

$$A = 15;$$

It is worth mentioning here that while the formal  $A$  in  $G$  is renamed, any call to  $G$  will still use the variable symbol  $A$ .

We can summarize the deglobalization process in two steps:

- (1) convert the globals into formals, and change all the calls to the function to incorporate the globals as actuals,
- (2) apply the formal parameter renaming rule,

For example, consider the following set of equations:

$$G(B) = A + B;$$

$$A = 15;$$

$$Z = F(3);$$

$$F(X) = G(X) + A;$$

According to the process of deglobalizing  $A$  in the definition of  $G$  we have

$$G(B,C) = C + B;$$

$$A = 15;$$

$$Z = F(3);$$

$$F(X) = G(X,A) + A;$$

Similarly, by following the procedure above, we deglobalize  $A$  in the definition of  $F$ . First, we convert  $A$  in  $F$  to a formal

$$F(X,A) = G(X,A) + A;$$

and change all calls to  $F$  to incorporate the new actual  $A$ , thus we write

$$Z = F(3,A) ;$$

Then we apply the formal parameter renaming rule and rename  $A$  in the definition of  $F$ . The definition of  $F$ , after renaming  $A$  as  $Y$  for example, becomes

$$F(X,Y) = G(X,Y) + Y ;$$

Therefore, the above set of Iswim-definitions are translated into

$$G(B,C) = C + B ;$$

$$A = 15 ;$$

$$Z = F(3,A) ;$$

$$F(X,Y) = G(X,Y) + Y ;$$

### 3.2.7- The Dilation Process:

It is worth emphasizing here that the deglobalization process described above does not work when the function is defined by a where-expression. That is, when the right hand side of the function definition is a where-expression. This is because the liquidation process may convert local variables to global ones.

Consider the following set of equations

$$B = F(3) ;$$

$$A = 10 ;$$

$$F(X) = V + A \text{ where}$$

$$V = A * A + B ;$$

$$B = 10 * X ;$$

end;

Let us start by converting the global  $A$  in the definition of  $F$  into a formal and



changing all the calls to  $F$  by incorporating  $A$ . The result is

$$B = F(3, A);$$

$$A = 10;$$

$$F(X, A) = V + A \text{ where}$$

$$V = A * A + B;$$

$$B = 10 * X;$$

$$\text{end;}$$

By renaming  $A$  in the definition of  $F$  as  $Y$ , and applying the renaming

$$\{B \leftarrow Z\}$$

to the inner clause we get

$$B = F(3, A);$$

$$A = 10;$$

$$F(X, Y) = V + Y \text{ where}$$

$$V = Y * Y + Z;$$

$$Z = 10 * X;$$

$$\text{end;}$$

If we liquidate the inner clause in the set of definitions then the locals  $Z$  and  $V$  will become globals. The way out here is to *dilate* the inner clause by successive application of the calling rule and the deletion rule. Thus, by calling  $V$  then deleting its definition from the body of the clause, the definition

$$F(X, Y) = V + Y \text{ where}$$

$$V = Y * Y + Z;$$

$$Z = 10 * X;$$

$$\text{end;}$$

becomes

$$F(X, Y) = (Y * Y + Z) + Y \text{ where}$$

$$Z = 10 * X;$$

$$\text{end;}$$

by repeating the process for  $Z$  we get

$$F(X,Y) = (Y*Y + (10*X)) + Y ;$$

in which the variables occurring in the definition are all formals.

The above set of definitions then becomes

$$B = F(3,A) ;$$

$$A = 10 ;$$

$$F(X,Y) = (Y*Y + (10*X)) + Y ;$$

Notice that the **dilation** process (successive calling and deleting) is equivalent to the amalgamation process. The only difference is that the amalgamation preserves the definitions of the inner clause, while the dilation discards them by incorporating them in the subject of the clause. The dilation process is more appropriate when we linearize function definitions because it preserves the scope of variables in the definition.

### 3.3- Functions Without Globals:

The target family of languages we propose in this thesis, *DE* (definitional equations) is an intensional language. It is intensional in the sense that the algebra of data types which determines a certain member of this family is intensional; also in the sense that the values of variables are dependent on the contexts. Hence, the intensional value of a symbol is a set of indexed objects, each is the value of the symbol at a certain world in the universe of the algebra. While the value of a constant symbol in the signature of the algebra is the same at each world, the value of a variable symbol varies from one world to another. The main question which needs to be answered in the compilation process of any source language is, what sort of intensional algebra we have to consider. More precisely, what the universe of the algebra should be and what the intensional operators we need the algebra to facilitate are.

Let us assume that we have the following two equations in *Iwade(Q)*:

$$F(A) = A * A + 2 ;$$

$$X = F(4) + F(5) ;$$

Clearly, the expressions  $F(4)$  and  $F(5)$  are not valid in *DE(Q)*. Neither is the definition  $F(A) = A * A + 2$ . Ideally, what we would like to be able to say instead, is

$$F = A * A + 2 ;$$

$$X = (F \text{ when } A \text{ has the value } 4) + (F \text{ when } A \text{ has the value } 5)$$

That is, we want to get rid of both function definitions and function applications.

In the above example, basically, we want the value of  $F$  at any world in the universe to be dependent on (a function of) the value of its formal  $A$  and the value of the numeral  $2$  at that same world. As the intensional algebra will be based on the pointwise extension of the algebra  $Q$ , then the constant symbols are all interpreted pointwise. Thus the values of  $*$ ,  $+$ , and  $2$  are the same in

every world. The only value then, which decides the value of  $F$  at a certain world is the value which the formal  $A$  takes at that same world. At the first call we want the value of  $A$  to be the value of 4, and at the second call it is the value of 5.

We propose to consider the universe of the intensional algebra to be the set of textual function calls. As the formal, then, takes different values from a function call to another the value of  $F$  changes accordingly. For the time being, let us denote this set by

$$U = \{ \langle v, i \rangle : v \text{ is a variable symbol and } i \in \mathbb{N} \}$$

Informally, an element  $\langle u, j \rangle$  in  $U$  denotes the  $j$ 'th call of the function symbol  $u$ . We emphasize here that  $\langle u, j \rangle$  is a purely syntactic notion; i.e. the pair  $\langle u, j \rangle$  represents the  $j$ 'th occurrence of the function symbol  $u$ .

Accordingly, we want the formal  $A$  in the above example to take the value 4 in the world represented by  $\langle F, 0 \rangle$ , and the value 5 in  $\langle F, 1 \rangle$ . Let us illustrate that by the following diagram

<div style="border: 1px solid black; padding: 5px; width: fit-content;"> the world <math>\langle F, 0 \rangle</math>  <math>A = 4</math>  <math>F = A * A + 2</math> </div>	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> The world <math>\langle F, 1 \rangle</math>  <math>A = 5</math>  <math>F = A * A + 2</math> </div>
---	---

Notice that the definition of  $F$  above is the same in the two worlds. However,  $A$  has two different definitions, each is valid in a certain world only. To represent such a change in the value of  $A$ , we introduce the operator **act** (for **actuals**) which forms a legitimate expression out of a list of expressions in the algebra. That is, if

$$x_0, \dots, x_{n-1}$$

is a list of expressions then so is

$$\text{act}(x_0, \dots, x_{n-1})$$

Informally, the value of the latter expression in the world  $\langle v, i \rangle$  is the value of the  $i$ 'th element of the list; i.e. it is the value of  $x_i$ . An important point to note



here is that, we are interested in the extensional value of  $x_i$ , and it is meaningless to talk about such a value without referring to a certain world in the universe. However, the actuals in our simple example are both constants, and the value of a constant is the same in all the worlds of the universe. We shall discuss the case when actuals are expressions containing variable symbols later in this section.

Thus we add to the target code of the program the equation

$$A = \text{act} (x_0, \dots, x_{n-1})$$

defining the formal  $A$  of the function  $F$ , where (for every  $i$ )  $x_i$  is the actual of the  $i$ 'th call of  $F$ . In our example we write

$$F = A * A + 2;$$

$$A = \text{act} (4, 5);$$

$$X = F(4) + F(5);$$

Lastly, since we compiled the non-nullary variable symbols ( $F$  in our example) to nullary ones, then function application is meaningless. However, the equation  $X = F(4) + F(5)$ , within the context of the above analysis, means that:

$$X = \text{the first call of } F + \text{the second call of } F;$$

For this, we introduce the family of intensional operators **call**. Each member of this family is indexed by a natural number, and  $\text{call}_i H$  for a natural number  $i$  and a function symbol  $H$ , denotes the  $i$ 'th call of  $H$ . Syntactically then, the equation

$$X = F(4) + F(5) \text{ is translated into } X = \text{call}_0 F + \text{call}_1 F;$$

and hence the two equations

$$F(A) = A * A + 2;$$

$$X = F(4) + F(5);$$

are translated into:

$$F = A * A + 2;$$

$$A = \text{act} (4, 5);$$

$$X = \text{call}_0 F + \text{call}_1 F;$$

So far we have considered a very simple case. A case where both actuals were constants. Moreover, we proposed the set of function calls as a universe of our intensional algebra. We take one step further now and assume that one of these actuals is a variable symbol. Consider the following three equations:

$$F(A) = A * A + 2;$$

$$X = F(Y) + F(5);$$

$$Y = 4;$$

According to the above analysis, we can compile them into the equations

$$F = A * A + 2;$$

$$A = \text{act} (Y, 5);$$

$$X = \text{call}_0 F + \text{call}_1 F;$$

$$Y = 4;$$

Suppose we want the value of  $\text{call}_0 F$  in the world represented by the pair  $\langle u, i \rangle$ , for a variable symbol  $u$  and integer  $i$ . (†) Ideally, the value of  $\text{call}_0 F$  in any world should be equivalent to the value of  $F$  in the world which differs from the present one, at most, in the value assigned to the formal  $A$ . In such a world, we want the value of  $A$  to be equal to the value of  $Y$ . We know that the world in the universe which satisfies such a condition is that represented by the pair  $\langle F, 0 \rangle$ . The value assigned to the symbol  $A$  here is the value of  $\text{act} (Y, 5)$  at  $\langle F, 0 \rangle$  which is equal to the value of  $Y$ . From the equation  $Y=4$  in the program,

---

† We would like to draw the attention here that we do not imply in the following discussion an operational semantics or any strategy for evaluating intensional expressions, our purpose at this stage is to discuss the structure of the universe and the effect of **act** and **call** on its elements.

we know that the value of  $Y$  is 4 at any world in the universe.

Thus, whatever the present world is, the intensional operator  $call$  causes a 'jump' from the present world to another depending on the argument of  $call$  and on its index. Moreover, the value of  $act(Y,5)$  in the world represented by  $\langle F,0 \rangle$  should be equal to the value of  $Y$  in  $\langle u,i \rangle$  where  $\langle u,i \rangle$  represents the world in which  $F$  was invoked. Thus, while  $call$  transfers the evaluation to another world in the universe, the intensional operator  $act$  retreats back to the world in which the last function was to be evaluated; i.e the calling world where the actuals are to be evaluated. We shall explain that formally later. Informally however, we can summarize the evaluation process as follows :

The value of  $call_1 F$  at  $\langle u,i \rangle$   
 is equivalent to the value of  $F$  at  $\langle F,0 \rangle$   
 which is equivalent to the value of  $A*A+2$  at  $\langle F,0 \rangle$   
 this depends on the value of  $A$  at  $\langle F,0 \rangle$   
 which is equal to the value of  $act(Y,5)$  at  $\langle F,0 \rangle$   
 and this should be equal to the value of  $Y$  at  $\langle u,i \rangle$   
 which is equal to the value 4  
 and thus,  $call_1 F$  at  $\langle u,i \rangle$  is  $4*4+2$

So far we have proposed to represent the universe as the set of function calls. However, there is an important point to be noted from the above analysis. We need to preserve the information that the first call of  $F$  was to be evaluated in  $\langle u,i \rangle$ . This is so that after  $call_0$  transfers the evaluation process to be at the world  $\langle F,0 \rangle$ , the operator  $act$  knows that  $F$  has been invoked from  $\langle u,i \rangle$ ; hence jumps back to such a world where the actuals should be evaluated. This is a crucial point specially when considering chained function calls; i.e functions invoking other functions and recursive function calls. In other words, we have to

preserve the list of calls occurred so far.

Clearly, we have to represent the universe of the algebra as lists of function calls rather than single ones. Hence, a point in our universe is a list whose head represents the present function call and whose tail is the list of function calls invoked so far and lead to the present one. Moreover, we don't need to incorporate the name of the function being called as this is uniquely determined by the list of calls which have been invoked so far. That is, our universe is the set of lists of natural numbers.

Within such a representation of the universe, the value of  $\text{call}_i F$  for a natural number  $i$  and a function symbol  $F$ , at a world  $u$  in the universe should be the value of  $F$  at the world whose head is the atom  $i$  and whose tail is the list  $u$ . Moreover, the value of

$\text{act}(x_0, \dots, x_{n-1})$  at a list  $[i j \dots]$

in the universe is the value of  $x_i$  at the list  $[j \dots]$ . Thus the effect of  $\text{call}$  and  $\text{act}$  on the worlds of the universe resembles 'stacking' and 'popping' in a stack structure. That is, while  $\text{call}$  puts an element as a head of the list,  $\text{act}$  pops out the head of the list.

Thus in the above example, the value of  $\text{call}_0 F$  at the list whose head is the natural number  $i$ , say  $[i \dots]$

is equivalent to the value of  $F$  at the list  $[0 i \dots]$

which is the value of  $A * A + 2$  at the list  $[0 i \dots]$

and this depends on the value of  $A$  at  $[0 i \dots]$ .

which equals the value of  $\text{act}(Y, 5)$  at  $[0 i \dots]$

which is equal to the value of  $Y$  at  $[i \dots]$

and this equals 4.

thus the value of  $\text{call}_0 F$  equals  $4 * 4 + 2$ .



### 3.4- The Family of Intensional Algebras *FUN*:

To formalize the discussion above we introduce the family of intensional algebras  $\{FUN(A) : A \text{ is an extensional algebra}\}$ . *FUN* is a function which maps extensional algebras to intensional ones and thus *FUN*(A), for an extensional algebra A, is uniquely determined by the algebra A.

#### Definition: The Intensional Algebra *FUN*

Given an extensional  $\Sigma$ -algebra A, *FUN*(A) is the intensional  $\Sigma'$ -algebra such that:

The signature  $\Sigma'$  of *FUN*(A) =  $\Sigma \cup \{\text{call}_i : i \in \omega\} \cup \{\text{act}\}$

The universe L of *FUN*(A) is the set of lists of natural numbers.

The extensional domain of *FUN*(A) is the domain of A.

*FUN* extends A pointwise upon the universe L, that is

for every n-ary constant symbol  $\psi$  in the signature of A,

for every n terms  $\xi_0, \dots, \xi_{n-1}$  in *FUN*(A),

and for every list  $\varphi$  in the universe L,

$$(FUN(A)(\psi(\xi_0, \dots, \xi_{n-1})))_{\varphi} = A(\psi)((FUN(\xi_0))_{\varphi}, \dots, (FUN(\xi_{n-1}))_{\varphi})$$

*FUN* assigns meaning to the family of constant symbols *call* as follows:

for every  $i \in \omega$

for every variable symbol  $\xi$ , and

for every list  $\varphi$  in the universe L

$$(FUN(A)(\text{call}_i(\xi)))_{\varphi} = \xi_{\text{cons}(i, \varphi)}$$

*FUN* assigns the following meaning to the constant symbol *act*

for every list  $\varphi$  in the universe  $L$

and for every  $n$  expressions  $\xi_0, \dots, \xi_{n-1}$

$$(FUN(A)(act(\xi_0, \dots, \xi_{n-1}))_\varphi = (\xi(hd(\varphi)))_{tl(\varphi)}$$

where  $hd$ ,  $tl$ , and  $cons$  are (resp.) the usual head, tail, and construct functions on lists.

According to the analysis above and the definition of  $FUN$ , we can compile the subfamily  $Iwade$  to equations over terms of the intensional algebra  $FUN$ . That is, for an extensional  $\Sigma$ -algebra  $A$ , the language  $Iwade(A)$  is compiled into the member  $DE(FUN(A))$ . The latter of course is a member in our target language  $DE$ .

### 3.5- Recursion and Nested Functions Calls:

In this section we show that the technique and the algebra we have already described does handle nested function calls and even recursion. As handling recursion is, typically, known to be more complicated than nested function calls, we give here a simple example showing how our technique deals with it. Consider the following  $Iwade(Z)$ -program:

**Fac (2) where**

**Fac(X) = if X le 1 then 1 else X \* Fac(X-1) fi;**

**end**

In the next section we shall give the algorithm for translating  $Iwade$ -programs into programs in  $DE$ . For the time being however, we shall use the informal analysis of the translation we described earlier. Using such an analysis, we translate this program into the following program in  $DE(FUN(Z))$

**result = call<sub>0</sub>Fac;**

**Fac = if X le 1 then 1 else X \*call<sub>1</sub>Fac fi;**

**X = act (2,X-1);**

Note that, the value of the *DE*-program should be the value of **result** at the empty list because no functions has been called yet.

The value of **result** at the list  $[]$  equals

the value of  $\text{call}_0 \text{ Fac}$  at  $[]$  (by substitution)

which is equal to the value of **Fac** at the list  $[0]$

(by the interpretation of **call**)

by direct substitution, this is equal to the value of

$(\text{if } X \leq 1 \text{ then } 1 \text{ else } X * \text{call}_1 \text{ Fac fi})$  at the list  $[0]$

By substitution, the value of **X** at  $[0]$  is equal to the value of

$\text{act}(2, X-1)$  at  $[0]$  which is 2 (by the interpretation of **act**)

.....(\*1)

Thus the value of **Fac** at  $[0]$  equals the value of

$(2 * \text{call}_1 \text{ Fac})$  at  $[0]$  which equals the value of

$2 * (\text{the value of Fac at the list } [1 \ 0])$

(note here that both **\*** and **2** are interpreted pointwise)

this (by substitution) is equal to

$2 * (\text{if } X \leq 1 \text{ then } 1 \text{ else } X * \text{call}_1 \text{ Fac fi})$  at  $[1 \ 0]$

now the value of **X** at  $[1 \ 0]$  is equal to the value of

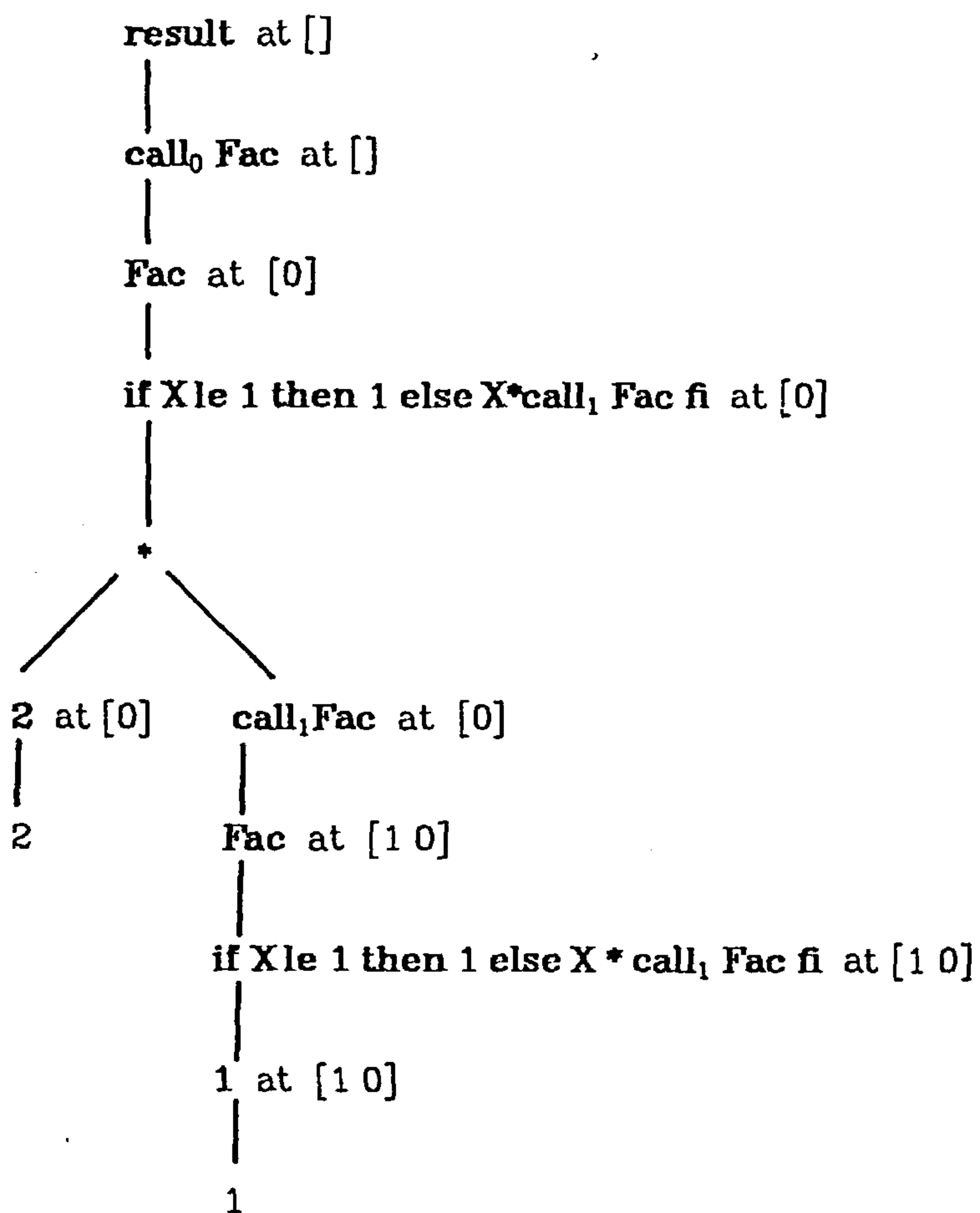
$\text{act}(2, X-1)$  at  $[1 \ 0]$  (by substitution)

which is the value of  $X-1$  at  $[0]$  (by the definition of **act**)

which is 1 (by (\*1) above, **X** at  $[0]$  is 2)

Hence, the value of **result** at  $[]$  is  $2 * 1$ .

The above evaluation can be represented in the following tree where lines denotes equalities:





### 3.6- From Iwade to DE:

#### 3.6.1- The Translation Algorithm

Given a program  $P$  in  $Iwade(A)$ , for an extensional algebra  $A$ , the target program  $Trans(P)$  in  $DE(FUN(A))$  is defined recursively as follows:

if  $P$  is a simple expression in  $Iwade(A)$  then  $Trans(P)$  is the singleton

$\{ \text{result} = P \}$

if  $P$  is of the form

$X \text{ where } D \text{ end}$

where  $X$  is a simple  $Iwade(A)$ -expression and  $D$

is a set of  $Iwade(A)$ -equations

then  $Trans(P)$  is

$\{ \text{result} = \text{compexp}_P(X) \} \cup \{ \text{compdef}_P(d) \mid d \in D \}$

where for a set of definitions  $C$ , a definition  $d$ , a valid expression  $\varepsilon$  and a where-expression  $\vartheta$

$\text{compexp}_\vartheta(\varepsilon) =$

if  $\varepsilon$  is of the form  $\text{op}(x_0, \dots, x_{n-1})$

where  $\text{op}$  is an  $n$ -ary constant symbol and

$x_0, \dots, x_{n-1}$  are  $n$  expressions, then

$\text{op}(\text{compexp}_\vartheta(x_0), \dots, \text{compexp}_\vartheta(x_{n-1}))$

if  $\varepsilon$  is a nullary variable symbol then  $\varepsilon$

if  $\varepsilon$  is of the form  $F(x_0, \dots, x_{n-1})$  where  $F$  is an

$n$ -ary variable symbol and  $x_0, \dots, x_{n-1}$

are  $n$  expressions, then

$\text{compfun}_\vartheta(F)$

$\text{compfun}_j(F) = \text{call}_i F$ .

where  $i$  is the number of times the function symbol  $F$   
has been applied so far. .... (see 3.6.2)

$\text{compdef}_j(d) =$

if  $d$  is of the form  $V = \varepsilon$ , where  $V$  is a nullary  
variable symbol, and  $\varepsilon$  is an expression in  
 $\text{Iwade}(A)$ , then

$\{V = \text{compexp}_j(\varepsilon)\}$

if  $d$  is of the form  $F(x_0, \dots, x_{n-1}) = \varepsilon$

where  $\varepsilon$  is a valid expression in  $\text{Iwade}(A)$ ,

$F$  is an  $n$ -ary variable symbol, and

$x_0, \dots, x_{n-1}$  is the list of formal parameters, then

$\{F = \text{compexp}_j \varepsilon\} \cup \{\text{compform}_{j,F}(x_i) \mid i \in n\}$

$\text{compform}_{j,F}(x) = \{x = \text{act}(\text{compexp}_j(a_0), \dots, \text{compexp}_j(a_{m-1}))\}$

where for each  $i$  in  $m$ ,  $a_i$  is the actual of

the  $i$ 'th invocation of  $F$  in  $\mathcal{V}$ . .... (see 3.6.2)

### 3.6.2- Comments on the Algorithm:

The way we defined the functions **compfun** and **compform** in the algorithm above is non-functional. This is, only, for the sake of simplicity. A functional algorithm can be easily written. The important thing which we have to observe in the algorithm is that, the  $i$ 'th occurrence of the function symbol  $F$ , will be translated into  $\text{call}_i F$ , and each actual of such an occurrence will be the  $i$ 'th element in the sequence of expressions defining the corresponding formal.

For example, if we assume that the following two equations are part of a program in which  $F(B,C,D)$  is the  $k$ 'th call of  $F$

$$F(X,Y,Z) = X + Y + Z;$$

...

$$A = F(B,C,D);$$

then,  $F(B,C,D)$  is translated into  $\text{call}_k F$

and the formals will be defined as

$$X = \text{act} (x_0, \dots, x_{k-1}, B, \dots)$$

$$Y = \text{act} (y_0, \dots, y_{k-1}, C, \dots)$$

$$Z = \text{act} (z_0, \dots, z_{k-1}, D, \dots)$$

Thus, making a correspondence between a function call and the actuals of such a call. Whether we count occurrences of a function call from left to right or from right to left, the main idea is to construct such a correspondence.

In the algorithm above, we assumed that counting calls was from left to right in an accumulative way just for the sake of simplicity. We preserved the correspondence between calls and their actuals by registering the actuals as well from left to right accumulatively. However, if we represent the expressions as lists, like in [Wad84], then writing and implementing a functional algorithm is straight forward. Counting the occurrences of calls to a certain function and registering the actuals of such calls can be done in a recursive manner by traversing the list representing the expression.

### 3.7- Compiling Functions with globals:

So far we have discussed the compilation of programs in which functions definitions do not have global variables. We considered the subset *Iwade* of *Iswim* as our source language; then defined the intensional algebra *FUN* and the language *DE(FUN)* as the target of the compilation.

In this section, we consider a bigger subset of *Iswim* than *Iwade*. We define the language *Ipaddle* to be the subset of *Iswim* which contains *Iwade* and allows global occurrences of nullary variable symbols in function definitions. The only constraint we put on *Ipaddle* is that it is not structured. In other words, *Ipaddle* is linear *Iswim*.

#### 3.7.1- The Language *Ipaddle*:

Given an extensional  $\Sigma$ -algebra *A*, the abstract syntax of *Ipaddle(A)* is the same as *Iwade(A)* except in the definition of the set of **valid definitions**. These are defined as follows:

A **valid definition** in *Ipaddle(A)* is of the form

$$\vartheta (\eta_0, \dots, \eta_{n-1}) = \varepsilon$$

where  $\vartheta$  is an *n*-ary variable symbol,

$\eta_0, \dots, \eta_{n-1}$  are *n* distinct and different from  $\vartheta$

nullary variable symbols,

and  $\varepsilon$  is a simple expression.

Notice that in defining the set of *Iwade(A)*-valid definitions, the variable symbols occur in the defining expression  $\varepsilon$  (the right hand side of the definition) were all formals in the definition. In *Ipaddle*,  $\varepsilon$  may contain global variable symbols; i.e not just formals.

Clearly, *Ipaddle* is a subfamily of the language *Iswim* and contains the family *Iwade*. For example, the definition



$$F(X,Y) = X + Y - B$$

is a valid Ipaddle-definition but not a valid definition in Iwade because  $B$  occurs as a global.

To transform Iswim-programs into programs in Ipaddle we use the program transformation rules described before except, of course, the deglobalization rule (the formal parameter rule).

### 3.7.2- The Compilation of Nullary Globals:

Consider the following program in Ipaddle

$F(3)$  where

$$F(X) = X + A;$$

$$A = Y + 4;$$

$$Y = 10;$$

end

Obviously, the occurrence of  $A$  in the definition of  $F$  is global, and so is the occurrence of  $Y$  in the definition of  $A$ . Since  $F(X)=X+A$ , we expect the value of  $F$  at any world to be a function of the value of  $A$  at that same world.

However, the value of the above program is the value of  $\text{call}_0 F$  at the empty list  $\Lambda$ .

This is (by the definition of  $\text{call}$  equivalent to the value of  $F$  at  $\text{cons}(0,\Lambda)$  .

As  $F$  is compiled into  $F=X+A$  and  $+$  is interpreted point wise,

then  $F$  at  $\text{cons}(0,\Lambda)$  is a function of the value of  $A$  at  $\text{cons}(0,\Lambda)$ . However,  $A$  is not defined at  $\text{cons}(0,\Lambda)$ . We expect  $A$  to be evaluated in the world  $\Lambda$ .

Notice that  $Y$  is global in the definition of  $A$ , however the value of  $A$  at any world is a function of the value of  $Y$  at that same world. Thus, even though  $A$  is global in the definition of  $F$  and  $Y$  is global in the definition of  $A$ , the two cases are different. This is because  $A$  occurs as a global in a function definition, while

$Y$  is a global in a nullary definition. Our interest is in the first case because switching worlds in the universe is due to function calls not nullary variables. The second case is straight forward and of no significance in the compilation process.

Since *Ipaddle* is a linear language, all the definitions in a program should occur in one textual level (where-clause). And thus, if a variable is global to a function definition (and not free in the program) then it should be defined in such a where-clause. Thus, it should be evaluated at the list representing the home world (the world of commencing evaluation); i.e. at the empty list  $\Lambda$ .

A very simple way to solve this problem is to add to the algebra *FUN*, the new operator symbol  $\gamma$ . In the above example, we compile the definition of  $F$  into

$$F = X + \gamma A$$

Informally,  $\gamma X$  means that  $A$  is global in the definition of  $F$ .

Formally, we extend the algebra *FUN* by adding the constant symbol  $\gamma$  to its signature. We define the meaning of  $\gamma$  as follows:

For any expression  $\varepsilon$ , and any list  $\vartheta$

$$(FUN(\gamma \varepsilon))_{\vartheta} = (FUN(\varepsilon))_{\Lambda}$$

In the above example then, the value of  $F$  at  $\text{cons}(0, \Lambda)$  is a function of  $\gamma X$  at  $\text{cons}(0, \Lambda)$ , which is the value of  $X$  at  $\Lambda$ .

### 3.7.3- Example:

In this section we demonstrate the compilation of programs in which function definitions have global occurrences of both nullary and function symbols. Consider the following program in *Ipaddle*:

**F(3) where**

**F(A) = G(A) + X;**

**X = 10;**

**G(C) = C + X;**

**end**

The nullary variable symbol  $X$  is global to both the definition of  $F$  and that of  $G$ . The function symbol  $G$ , also, is a global in the definition of  $F$ . Let us assume that we are using the algebra *FUN* (together with the intensional operator  $\gamma$ ). According to our previous analysis, the above program can be translated into

**result = call<sub>0</sub>F;**

**F = call<sub>0</sub>G +  $\gamma$  X;**

**X = 10;**

**G = C +  $\gamma$  X;**

**A = act (3);**

**C = act (A);**

The value of the program is the value of **result** at the empty list  $\Lambda$ , which is the value of **call<sub>0</sub>F** at  $\Lambda$ .

This is equivalent to the value of **F** at the list  $[0]$

which is the value of **call<sub>0</sub>G +  $\gamma$  X** at  $[0]$

Now the value of  $\gamma X$  at  $[0]$ , according to the definition of  $\gamma$  described before, is the value of  $X$  at  $\Lambda$ , which is 10.

On the other hand, the value of **call<sub>0</sub>G** at  $[0]$

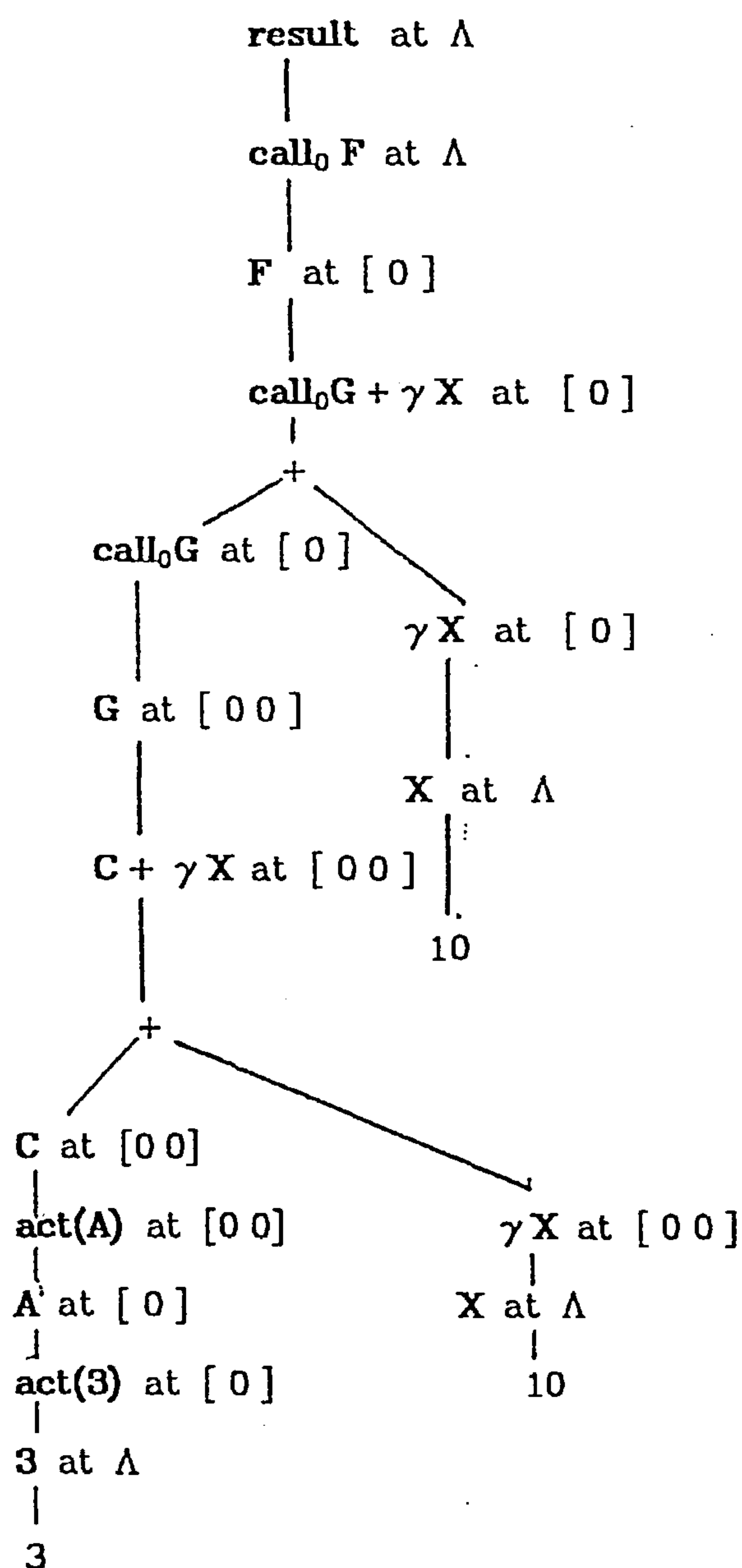
is the value of **G** at  $[0\ 0]$

which is the value of **C +  $\gamma$  X** at  $[0\ 0]$

The value of  $\gamma X$  at  $[0\ 0]$ , according to the definition of  $\gamma$ , is the value of  $X$  at  $\Lambda$ .

As we mentioned in the previous chapter, the effect of `call` and `act` resembles the functions 'push' and 'pop' on a stack. While `call` pushes the new world representing the recent function call, `act` retreats back to the calling world to evaluate the actuals of the function being called. At any world, however, the operator  $\gamma$  switches the evaluation to the home world represented by the empty list  $\Lambda$ .

The evaluation of the above example then can be represented by the following tree in which lines represent equalities:





## Chapter 4

### The Compilation of Iswim

#### 4.0- Introduction:

The languages we have discussed and compiled in chapter 3 are all linear functional languages. Linear means that programs in these languages are not modular or structured. In this chapter we shall discuss the compilation of structured functional languages into our target equational intensional language *DE*. The source language we shall consider is the language Iswim [AsWa80, WaAs84]. We want to draw the attention to the fact that Iswim here is the variant, defined by E.Ashcroft and W.Wadge, of Landin's ISWIM. Thus it is a first order functional language (see chapter 0).

Besides being an example of a structured functional language, our interest in Iswim comes from the fact that it is the basis of Luswim and hence of Lucid. We recall here that Lucid is Luswim with nested iteration (or freezing), where Luswim is the family of languages

$\{\text{Iswim}(\mathcal{L}_A(A)) : A \text{ is an extensional algebra}\}.$

The compilation of Iswim will consist of two main procedures:

first: We define the language  $\text{Iswim}^\dagger$ , which is the subset of Iswim in which the where-clause is allowed only in defining functions. Thus Iswim is still a structured language but simpler than Iswim. Programs in Iswim are transformed to equivalent ones in Iswim by the set of program manipulation rules described before.

next: We show, by means of examples, the complexities associated with

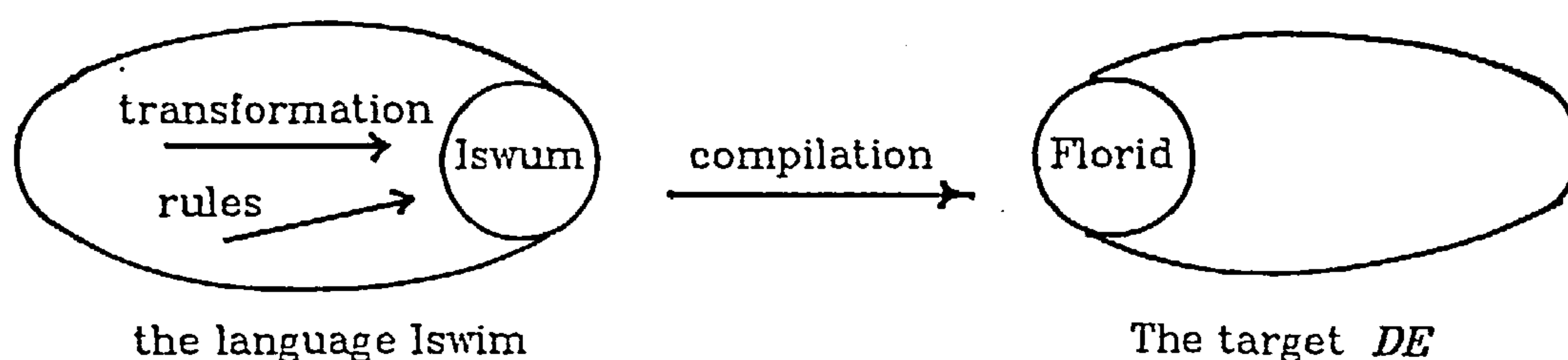
---

† I See What U Mean.

compiling structured functional languages, and show that the algebra *FUN* (defined in chapter 3) is not sufficient as a target algebra for the compilation. Instead, we define the family of intensional algebras *Flo*. Given an extensional algebra *A*, the language *Iswum*(*A*) is compiled into the target language *DE*(*Flo*(*A*)). We call the family of intensional equational languages *DE*(*Flo*) the family *Florid*. That is,

$$\text{Florid} = \{ DE(Flo(A)) : A \text{ is an extensional algebra} \}$$

*Florid*, then, is the target for compiling the language *Iswum*, and hence the target for *Iswim*. Thus, the procedure of compiling *Iswim* can be illustrated in the following figure



In the next chapter, we discuss two techniques for evaluating programs in *Florid*. The first is reductive and is based on a set of rewrite rules for the algebra *Flo*; and the second is eductive and is based on the idea of *eduction* described before.

#### 4.1- The Language *Iswum*:

We define here the family of structured languages *Iswum*. It is the subfamily of *Iswim* which:

- (1) Allows the **where**-clause expression only in function definitions. For example, the following is an *Iswum*-program

```

F(2,X) where
  X = 2;
  F(A,B) = Z + V where
    Z = A * A;
    V = 2 * B;
  end;
end

```

However, the lswim-expression

```

F(2,X) where
  F(A,B) = A * A + 2 * B;
  X = Z + V where
    V = 12;
    Z = V * V;
  end;
end

```

is not a program in lswum because the inner where-clause is used to define the nullary variable symbol X

(2) Programs in lswum are compatible; that is a variable symbol is bound at most once in the whole program. The following, for example, is an lswim-expression but not a legitimate expression in lswum as the variable symbol A occurs as a local in the main clause and a formal in the definition of F.

```

F(3)+A where
  F(A) = A * A;
  A = 4;
end

```

We introduce here the abstract syntax of lswum. Then using the program transformation rules described in chapter 3, we describe by means of examples the transformation of lswim-programs into equivalent programs in lswum.

#### 4.1.1- The Abstract Syntax of lswum:

Given an extensional  $\Sigma$ -algebra A, a program in lswum(A) is a valid expression. A valid expression is either simple or a where-clause, and these are defined recursively as follows:

- 1- If  $\alpha$  is either an  $n$ -ary constant symbol in  $\Sigma$ ,  
or an  $n$ -ary variable symbol; and  
 $x_0, \dots, x_{n-1}$  are simple expressions,  
then  $\alpha(x_0, \dots, x_{n-1})$  is a simple expression.
- 2- If  $\varepsilon$  is a simple expression and  $\delta_0, \dots, \delta_{m-1}$   
are  $m$  compatible valid definitions, then

$\varepsilon$  where

$\delta_0$

...

$\delta_{m-1}$

end

is a where-clause.

A valid definition is defined as

- 1- If  $\vartheta$  is a nullary variable symbol and  $\varepsilon$  is a simple  
expression. Then

$\vartheta = \varepsilon$  is a valid definition.

- 2- If  $\vartheta$  is an  $n$ -ary variable symbol,

$\eta_0, \dots, \eta_{n-1}$  are  $n$  distinct and different from  $\vartheta$   
nullary variable symbols,

and  $\varepsilon$  is a valid Iswum-expression (simple or a where-clause),

then

$\vartheta(\eta_0, \dots, \eta_{n-1}) = \varepsilon$  is a valid definition.

The compatibility of a set of definitions in Iswum is exactly the same as that for Iwade and Ipaddle. That is, a variable symbol occurs as the left hand side of an equation or as a formal parameter in a function definition only once in the whole program.



Notice that, apart from the compatibility requirement, the only difference between *Iswim* and *Iswum* is in the definition of the set of **valid definitions** of each. In the latter, if the left hand side of a definition is a nullary symbol then the right hand side expression should be simple (cannot be a where clause).

We define the semantics of  $\text{Iswum}(A)$  to be that of *Iswim*, i.e. the meaning of a program in  $\text{Iswum}(A)$ , for an algebra  $A$ , is the same as its meaning according to the semantics of *Iswim*.

#### 4.2- From *Iswim* to *Iswum*:

While a variable symbol must be bound at most once in an *Iswum*-program, a variable symbol can occur as a local in two different where-clauses in *Iswim*. In this case, clearly, we apply the renaming rule and transform the *Iswim*-program so that each variable symbol is bound at most once, and thus satisfy the compatibility requirement for *Iswum*-programs.

The other difference between *Iswum* and *Iswim* is that the latter allows the where-clause expression to occur in the right hand side of any definition; in the former however, it is only allowed in expressions defining functions. The transformation process, here, is straight forward by using the amalgamation rule described before. We give here a simple example to show the two cases.

**Example** Consider the following program in *Iswim*

```

F(2,X) where
  F(A,B) = A*A + 2*B;
  X = Z + A where
    A = 12;
    Z = V*V;
  end;
end

```

The variable symbol  $A$  occurs as a formal parameter to the definition of the function  $F$  and as a local in the where-clause defining the nullary symbol  $X$ . Of course, we can apply the amalgamation rule and amalgamate the inner where-

clause into the main one. However, this may work for this example only because  $A$  is a formal for the function variable  $F$ . If  $A$  is bound in the outer clause by a definition then amalgamation will result in two definitions for the variable  $A$ . Renaming, usually, should precede the amalgamation process. In the example above then, applying the renaming  $\{ V \leftarrow A \}$  (for example) on the inner where-clause, followed by the amalgamation rule we transform the program into

```

F(2,X) where
  F(A,B) = A*A + 2*B;
  X = Z + V;
  V = 12;
  Z = V*V;
end

```

#### 4.3- The Import Rule for Globals:

Our interest in this thesis, and in this chapter in particular, is mainly with the intensional approach to compiling globals and this will be discussed thoroughly in the coming sections. However, it is worth mentioning here that allowing structured function definitions in *lswum* enables us to get rid of globals in function definitions by importing them to the expression defining the function. The **import rule** [WaAs84] states that, any definition in the body of a where-clause can be added to any expression appearing in the right hand side of any definition in its body provided that such an importation is safe. An importation is safe when it preserves the compatibility of the expression; that is, when the left hand side of the imported definition is neither local in the expression nor a formal of the importing definition (the definition whose right hand side is the importing expression).

For example, importing the definition of  $V$  or that of  $X$  to the inner where clause in the following program is not safe; this is because  $V$  is local in the inner where-clause expression and  $X$  is a formal in the definition of  $F$ .

```

F(X,A) where
  X = A+Z;
  V = W+3;
  F(X,Y) = X*X + 2*V where
    V = Y*Y + 2*X + 5*A;
  end;
  A = Z * Z;
end

```

However, we can import the definition of A to the inner clause and obtain

```

F(X,A) where
  X = A+Z;
  V = W+3;
  F(X,Y) = X*X + 2*V where
    V = Y*Y + 2*X + 5*A;
    A = Z * Z;
  end;
  A = Z * Z;
end

```

Using the import rule, we can get rid of all global variables in function definitions by importing them to the expression defining the function symbol. Simply, if the importation is not safe then we can apply the renaming rules and ensure the safety of the importation.

In the following program, for example, we import the definitions of both G and Z into the expression defining F.

```

F(Z) where
  F(A) = A*A + G(Z) + Z;
  Z = 10;
  G(Y) = Y*2;
end

```

is transformed into

```

F(Z) where
  F(A) = A*A + G(Z) + Z where
    Z = 10;
    G(Y) = Y * 2;
  end;
  Z = 10;
  G(Y) = Y * 2;
end

```

Notice that the new program is in lswim but not a legitimate program in lswum as each of the symbols  $G$  and  $Z$  is defined twice. However, applying the renaming  $\{V \leftarrow Z, W \leftarrow G\}$  (for example) on the inner where-clause expression will yield the lswum-program

```

F(Z) where
  F(A) = A*A + W(V) + V where
    V = 10;
    W(Y) = Y * 2;
    end;
  Z = 10;
  G(Y) = Y * 2;
end

```



#### 4.4- What is Peculiar to Structured Languages:

Because *Ipaddle* is a linear language (non structured) any global to a function definition should be defined in the main, and only, where-clause in the program. Thus enriching the intensional algebra *FUN* by the operator symbol  $\gamma$  was enough to solve the problem, see section 3.7. The operator  $\gamma$  was defined as

$$(FUN(A)(\gamma \varepsilon))_{\mathfrak{A}} = (FUN(A)(\varepsilon))_{\Lambda}$$

where  $A$  is an extensional algebra,  $\varepsilon$  is an *Ipaddle*-expression, and  $\mathfrak{A}$  is a list of natural numbers. Notice also that  $\Lambda$  is the empty list representing the world in which no function has been invoked yet. Thus, no matter which world we are in at a certain instance in the evaluation, the operator  $\gamma$  will take us back to the world in which the global is defined and hence should be evaluated in.

In *Iswum*, the case is different. A global to a function definition does not always mean that it is defined at the outer most clause. For example, in the expression

```

F(3) where
  F(A) = G(A) + Y where
    G(B) = C + B;
    C = 5 ;
    end;
  Y = 10;
end

```

$Y$  is global in the definition of  $F$  and  $C$  is global in the definition of  $G$ . However, while  $Y$  is defined in the outer most where-clause,  $C$  is defined in the inner one. Therefore, if we compile

$$G(B) = C + B \quad \text{into} \quad G = \gamma C + B$$

and attempt to evaluate  $\gamma C$  using the definition of  $\gamma$  as given by the algebra *FUN*, then we end up attempting to evaluate  $C_{\Lambda}$  which is undefined.

In this particular example, the global variable  $C$  is defined in the same where-clause as the definition of  $G$ . An alternative solution to the problem, then,

would be to interpret  $\gamma$  as

$$(\gamma \varepsilon)_v = \varepsilon_{\text{tail}(v)}$$

where  $\varepsilon$  is an lswum-expression and  $v$  is a list. That is, to define  $\gamma$  such that it retreats back to the previous world or the world which invoked the present function call.

Clearly, this will not work in general. For example, if the definition of  $C$  in the expression above is in the outer where-clause then we need to go back two worlds from the present one. That is, if we are in the world represented by the list  $v$  then we need to go back to that represented by  $\text{tail}(\text{tail}(v))$ . This can be overcome, however, by compiling the definition of  $G$  into

$$G = \gamma \gamma C + B$$

and interpret  $\gamma\gamma$  as

$$\text{for any list } v \text{ and any expression } \varepsilon, \quad (\gamma\gamma \varepsilon)_v = (\varepsilon)_{\text{tail}(\text{tail}(v))}$$

This is not the only problem in this context. In the above example  $G$  is local to the definition of  $F$ ; let us consider a program in which  $G$  is global to  $F$ :

```
F(3) where
  F(A) = G(A) + Y;
  X = 10;
  G(C) = C + X;
end
```

Assume that we are using the algebra *FUN* together with the intensional operator  $\gamma$  defined above. According to our previous analysis, the above program can be translated into

```
result = call0F;
```

```
F = call0G +  $\gamma$  X;
```

```
X = 10;
```

```
G = C +  $\gamma$  X;
```

```
A = act (3);
```

```
C = act (A);
```

The value of the program is the value of `result` at the empty list  $\Lambda$ , which is the value of `call0 F` at  $\Lambda$ . This is equivalent to the value of `F` at the list `[0]`

which is the value of `call0G +  $\gamma$  X` at `[0]`

Now the value of  `$\gamma$  X` at `[0]`, according to the definition of  `$\gamma$`  described before, is the value of `X` at  $\Lambda$ , which is 10.

On the other hand, the value of `call0G` at `[0]`

is the value of `G` at `[0 0]`

which is the value of `C +  $\gamma$  X` at `[0 0]`

The value of  `$\gamma$  X` at `[0 0]`, according to the definition of  `$\gamma$` , is equivalent to the value of `X` at `[0]`. However, the value we need here is that of `X` at  $\Lambda$ .

Notice that `X` occurs as a global in the definitions of both `F` and `G`; however, switching the worlds in the universe, because of `call`, has resulted in evaluating `X` at two different worlds.

The above definition of  `$\gamma$`  is equal to that of `act`, i.e. it evaluates its argument in the world representing the latest function call. However, as `G` occurs as a global in the definition of `F`, we ended up jumping two function calls but only one  `$\gamma$`  (one world back). Thus, it is not enough to know that the occurrence of `X` in the definition of `G` is global. We need to know, as well, the place where `G` itself is defined so that we bind the global occurrences in its definition to their proper values.

Therefore, we need a more elaborate universe than the set of lists. We need a structure which registers the subsequent function calls and at the same time points out the `place` of each function definition so that we bind the globals to their definitions properly.

#### 4.5- Lists with Back Pointers:

The universe of the algebra *FUN* described in chapter 3 was the set of lists of natural numbers. The head of a list represents the present function call while the tail represents the sequence of calls which lead to the present one.

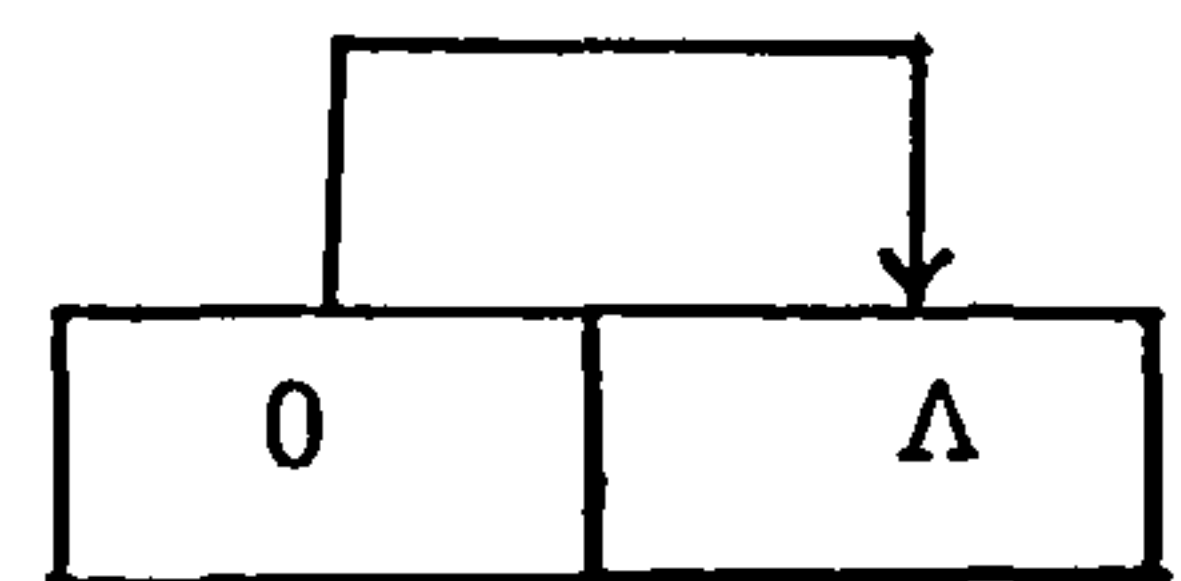
In the previous section however, we have noticed that in order to compile the language *lswum* we need a more elaborate universe for the target algebra. Beside knowing the list of function calls which lead to the present call, we need to know the **place** where the present function is defined.

In this section, we propose a new type of lists, where each list has two tails, the **dynamic tail** (or the **calling tail**), and the **static tail** (or the **defining tail**). To distinguish between these lists and the usual 'McCarthyian' lists, we shall call them **b-lists**, or **lists with back pointers**.

For example, in the program

```
F(3) where
  F(A) = G(A) + X;
  X = 10;
  G(C) = C + X;
end
```

described above, *F* should be evaluated at the b-list whose head is 0 and whose dynamic tail is equal to its static tail which is the empty list. This is because, *F* is both defined and called at the outermost level when no function has been either called or defined yet. It is evaluated at the b-list represented by the opposite diagram where the back



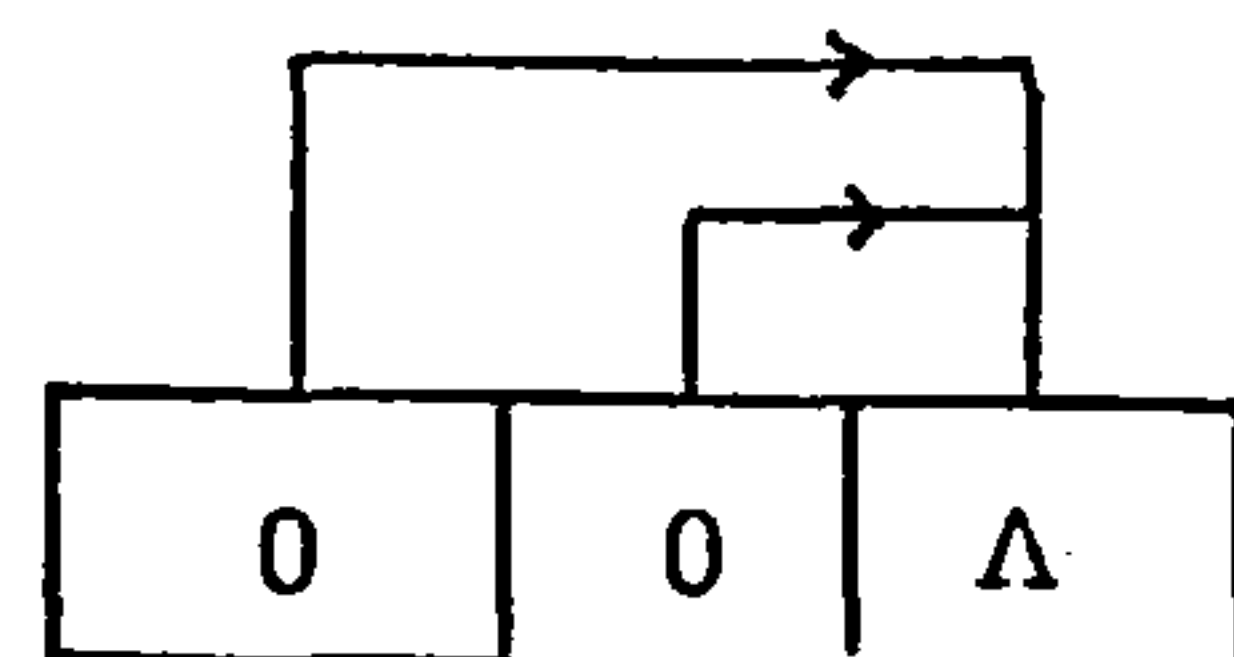
pointer† points at the b-list *Λ* which represents the world of commencing the evaluation (i.e. the world at which no functions had been defined or invoked yet).

---

† we are prompted to call them back pointers because the pointers originate from the head and point backwards towards the tail.



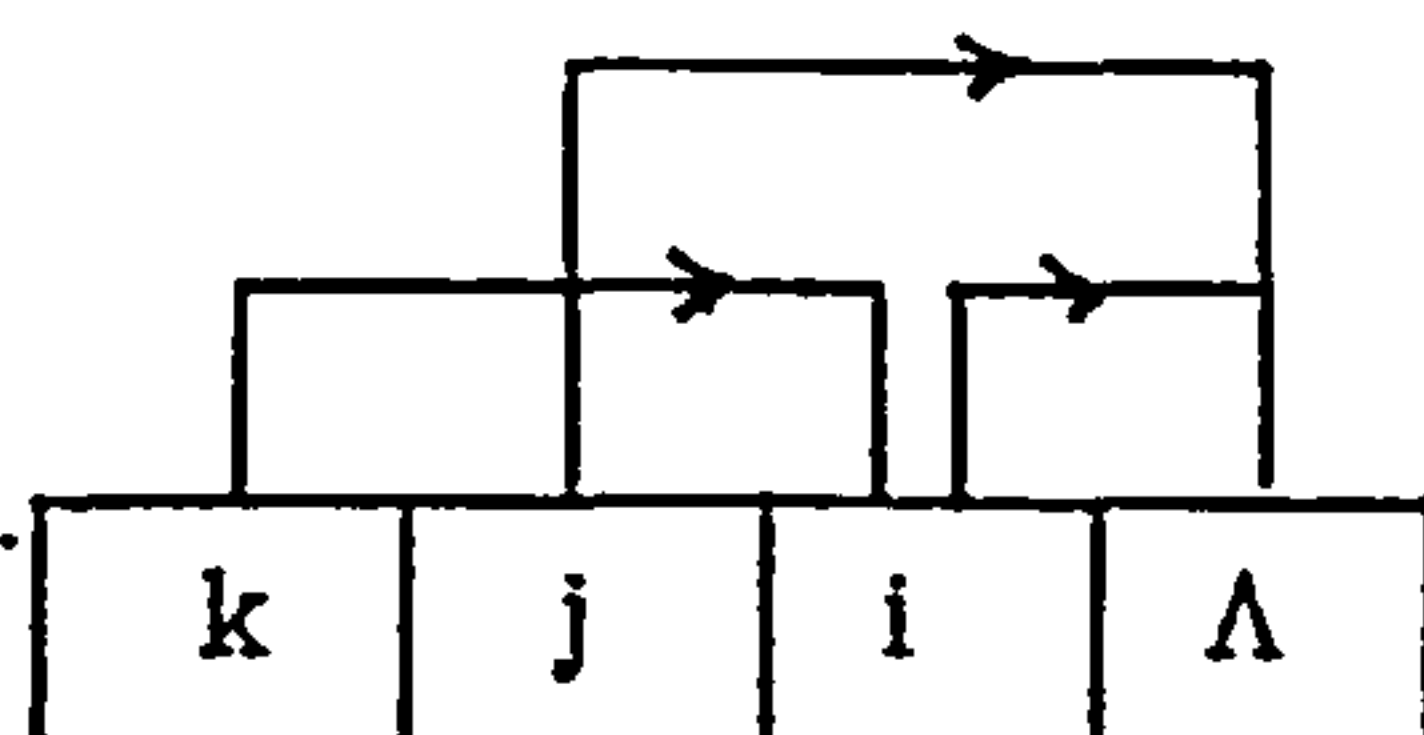
In the same example however, as  $G$  is defined at the outermost level and called within the first call of  $F$  it should be evaluated at the b-list whose head is 0 (as it is the first call of  $G$ ), and whose dynamic tail is the b-list represented by the above diagram (because it is called within the first call of  $F$ ). The static tail or the defining tail of the list should be the empty b-list. That is,  $G$  should be evaluated at the b-list represented by the opposite diagram



Consequently, the value of the global  $X$  should be taken at the defining tail, or the static tail, of the b-list not at the calling tail. That is, we have to evaluate  $X$  at the b-list nil, and this according to our program is equal to the value 10.

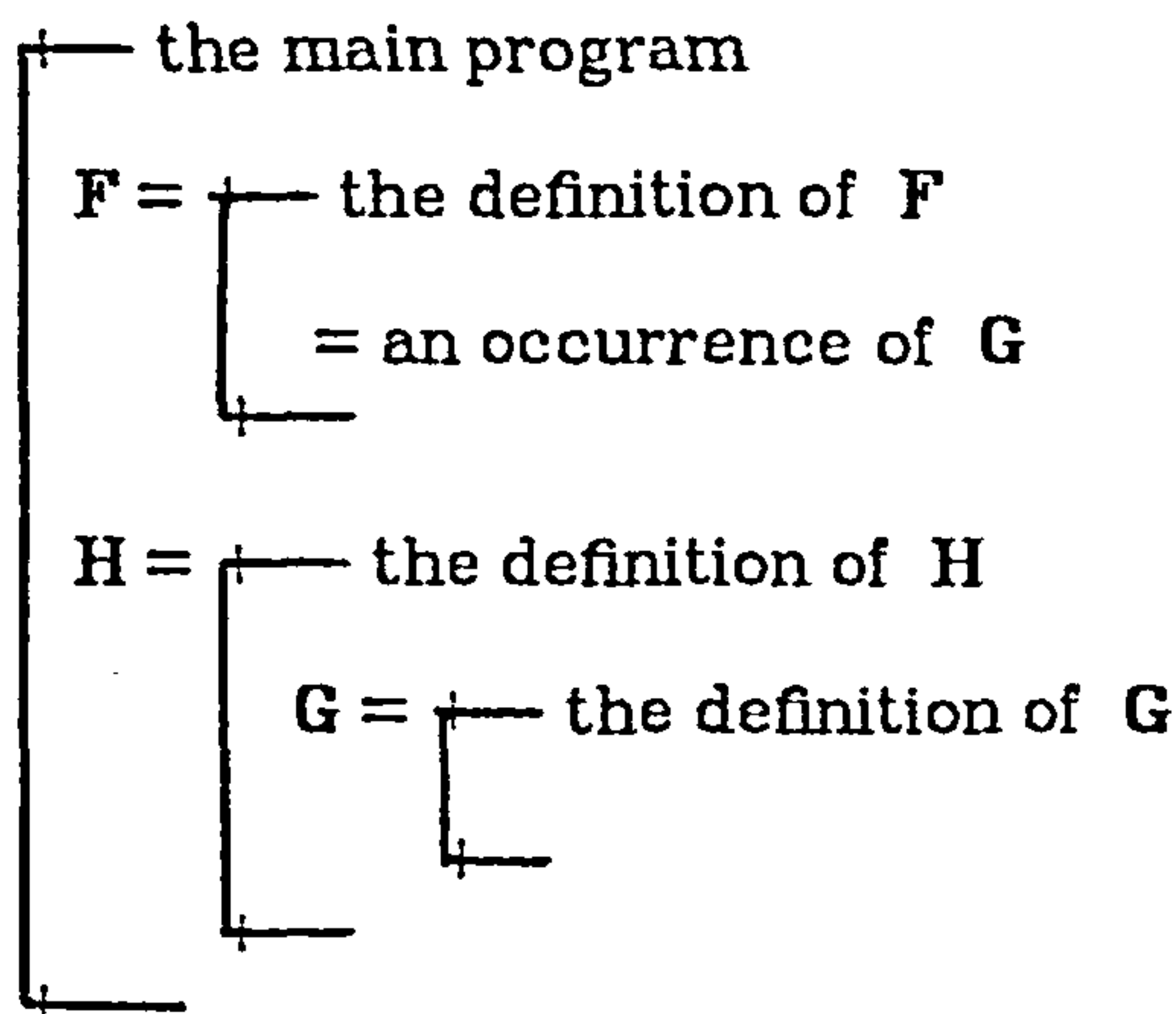
Therefore, if we define the operator  $stl$ , for static tail, on these lists then the value of  $\gamma \varepsilon$  for any expression  $\varepsilon$  at any b-list  $\varphi$  should be equal to the value of  $\varepsilon$  at the b-list  $stl(\varphi)$

It is important to stress on the fact that the static tail of a b-list should be an **initial segment** of its dynamic tail. This, in diagrams, means that a back pointer of a b-list always points at a previous cell (or atom) without crossing over any previous back pointer. This is important in the case of compiling structured languages in particular, like *Isdim*. For example, the following diagram does not represent a b-list, because the pointer referring to its static tail crosses over the pointer which is coming out of the cell  $j$ .



In reality, this means that (for some natural numbers  $k, j$ ) the  $k$ 'th call of a function  $G$  is invoked by the  $j$ 'th call of a function  $F$ . At the same time  $G$  is defined within the definition of a function  $H$ , and both  $H$  and  $F$  are defined in

the same where-clause. This exactly corresponds to the following structured program



Note that  $G$  is defined within  $H$ , and since scope propagates inwards, the occurrence of  $G$  in the definition of  $F$  should not refer to that inside  $H$ . Having a cross over, in the diagram, means that a function has been called outside its scope which is against our intuitive ideas.

We would like to draw the attention that these lists can be formalized as special type of binary trees. A node in a tree represents the head of the b-list (or the present function call) while the two offshoots represent the tails of the b-list. Moreover, the reader should note that such type of lists can be thought of as a formal representation of the block structure mechanism of D.Gries [Gri71].

We give now the formal definition of the set of b-lists, or **lists with back pointers** over an arbitrary set; together with the partial order **initial segment** defined on the elements of this set. We define, also, the operators **hd**, **dtl**, and **stl** for **head**, **dynamic tail** and **static tail** respectively. Then we define the b-lists constructor **link** which takes an atom together with two b-lists and forms the b-list whose head is the atom, dynamic tail is the first b-list, and static tail is the second.

**Definition:** Let  $A$  be a set. We define the set of b-lists  $bl(A)$ , the set of lists with back pointers over the set  $A$ , simultaneously with the relation initial segment, denoted by  $\prec$ , on the elements of  $bl(A)$  as follows:

1-  $\Lambda$  is in  $bl(A)$ , and  $\forall \alpha \in bl(A) \Lambda \prec \alpha$

(we call  $\Lambda$  the empty b-list)

2- if  $a \in A$ ;  $c, d \in bl(A)$  then

$\alpha = \langle a, c, d \rangle \in bl(A)$  whenever  $d \prec c$ .

3-  $\forall \alpha = \langle a, c, d \rangle$  and  $\beta = \langle a', c', d' \rangle$  in  $bl(A)$

$\alpha \prec \beta \Leftrightarrow \alpha = \beta$  or  $\alpha \prec d'$

**Definition:** We define the following functions on the non-empty elements of  $bl(A)$ ; i.e on the elements of  $bl(A)$  except  $\Lambda$

for any  $\alpha = \langle a, c, d \rangle \in bl(A)$

$hd(\alpha) = a$ ;  $dtl(\alpha) = c$ ;  $stl(\alpha) = d$

**Definition:** We define the b-lists constructor **link**, which takes an element  $a \in A$ , and two lists  $\alpha, \beta \in bl(A)$ , where  $\beta \prec \alpha$ , and constructs the new b-list  $\langle a, \alpha, \beta \rangle$  whose head is  $a$ , dtl is  $\alpha$ , and stl is  $\beta$ .

For any  $a \in A$ , and  $\alpha, \beta \in bl(A)$  where  $\beta \prec \alpha$ , the new b-list **link**( $a, \alpha, \beta$ ) is the b-list which satisfies the following equalities

$hd(\text{link}(a, \alpha, \beta)) = a$

$dtl(\text{link}(a, \alpha, \beta)) = \alpha$

$stl(\text{link}(a, \alpha, \beta)) = \beta$

#### 4.6- The Intensional Algebra *Flo* and the Language Florid:

In the previous sections, we have proposed the set of lists with back pointers over the natural numbers to be the universe for compiling structured languages. In this section we are going to define the family of intensional algebras *Flo* which will be the target algebra for compiling the source language *Iswum*. For a data algebra *A*, the language *Iswum*(*A*) will be compiled into *DE*(*Flo*(*A*)). The family of intensional equational languages *DE*(*Flo*) will be called *Florid*.

Before giving the formal definition of *Flo* we would like to cast some light on the intensional operators we need *Flo* to facilitate.

##### 4.6.1- Intensional Operators in *Flo*

In this section we shall discuss some of the intensional operators of the algebra *Flo*. Consider the following linear program in *Iswum*. Notice that the program is also an *Ipaddle*-program and can be compiled using the algebra *FUN* and the technique described in chapter 3; however, our interest here is mainly with the algebra *Flo*.

*Z* where

$$Z = F(3) + X;$$

$$F(A) = G(A) + X;$$

$$G(C) = C + X;$$

$$X = 10;$$

end

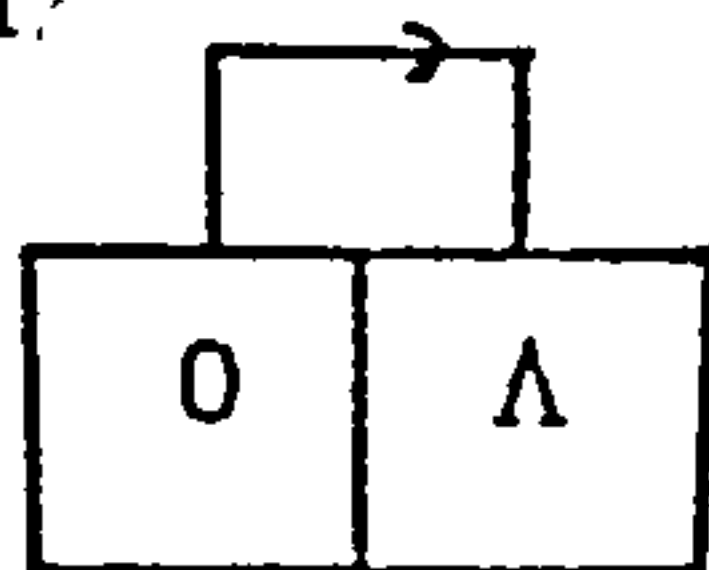
The value of the above program is the value of *Z* at the empty b-list  $\Lambda$  because there has not been any function call yet.

By direct substitution, this is equal to the value of the expression  $F(3)+X$  at the b-list  $\Lambda$  and this should be a function of the value of  $F(3)$  at  $\Lambda$ .

As  $F(3)$  is the first call to *F* and there has not been any function call yet, then



$F$  should be evaluated at the b-list whose head is 0 and whose dynamic tail is  $\Lambda$ . Moreover, as  $F$  is called in an expression defining a nullary variable symbol, and it is local in the set of definitions, then the static tail of the b-list should be, also,  $\Lambda$ . Therefore,  $F$  should be evaluated at the b-list, say  $\alpha$ , represented by the diagram:



the b-list  $\alpha$

Thus, for local function calls like  $F$  above, we introduce the family of operator symbols  $\text{call}$ ; it is the family  $\{\text{call}_i : i \in \omega\}$ . In the compilation, given an Iswum-program  $P$ , if the  $i$ 'th occurrence of a function symbol  $H$  in  $P$  is either local to an expression or global to an expression defining a nullary global symbol then such an occurrence is compiled into  $\text{call}_i H$ . In the above program, for example, we write

$\text{result} = Z$

$Z = \text{call}_0 F + X$

Formally, we define the operator  $\text{call}$  as follows

for every  $\varphi \in \text{bl}(\omega)$

for every expression  $\varepsilon$

and for every natural number  $i$

$$(\text{call}_i \varepsilon)_\varphi = (\varepsilon)_{\text{link}(i, \varphi, \varphi)}$$

Moreover, we introduce the operator symbol  $\text{act}$  (for actuals). If a function symbol whose formal is  $b$  (assuming for simplicity that the function is unary) occurs  $n$  times in the program with actuals

$x_0, \dots, x_{n-1}$ , then using the symbol  $\text{act}$  we add to the target program a new definition defining the list of actuals of the function by writing

$$b = \text{act}(x_0, \dots, x_{n-1})$$

where  $x_i$ , for any  $i$ , is the actual of the  $i$ 'th occurrence of  $F$ .

In the above program, The function  $F$  has been called only once in the whole program, so we add to the target program the definition

$A = \text{act}(3)$

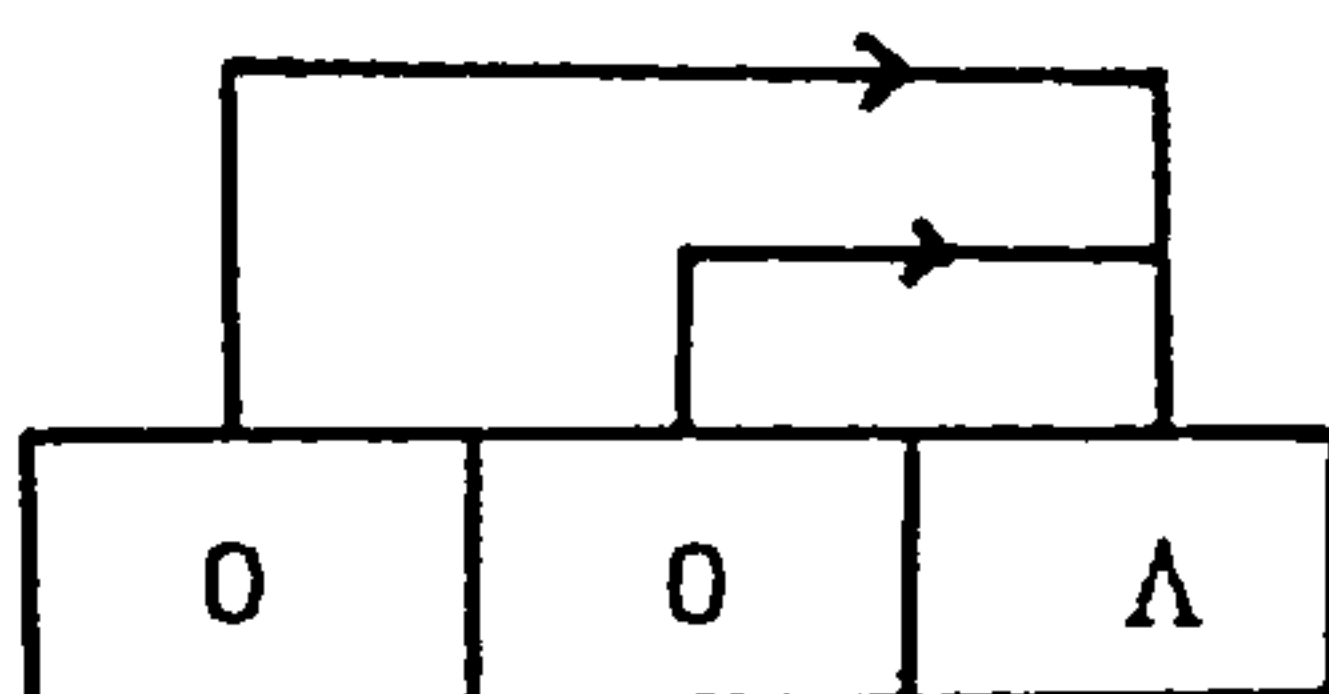
Formally, we define the operator  $\text{act}$  as follows:

For every b-list  $\varphi$ , and a sequence of expressions  $X_0, \dots, X_{n-1}$

$$(\text{act}(X_0, \dots, X_{n-1}))_{\varphi} = (X_{\text{hd}(\varphi)})_{\text{dtl}(\varphi)}$$

Notice that the operator  $\text{act}$  switches the evaluation to the dynamic tail of the present b-list, because it is the b-list which represent the sequence of function calls which lead to the present one.

In the above example, a call to  $F$  will invoke a call to  $G$ . As  $G(A)$  is the first call to  $G$  and invoked by the first call to  $F$  then  $G$  should be evaluated at the b-list whose head is  $0$  and whose dynamic tail is the b-list  $\alpha$  described above. The static tail of such a list, however, should be  $\Lambda$  because  $G$  is defined at the outer most level. Hence,  $G$  should be evaluated at the b-list, say  $\beta$ , represented by the diagram



the b-list  $\beta = \text{link}(0, (\text{link}(0, \Lambda, \Lambda)), \Lambda)$

What matters when we translate such a call to  $G$  is to state the fact that,  $G$  is called from the present world but its globals should be evaluated in the previous one. For this purpose, we introduce the family of operator symbols  $\text{gcall}$ . Like  $\text{call}$ ,  $\text{gcall}$  is the family  $\{\text{gcall}_i; i \in \omega\}$ .

Informally speaking, given a linear lswum-program  $P$ , if the  $i$ 'th occurrence of a function symbol  $H$  is global to an expression  $\varepsilon$  and  $\varepsilon$  is not the right hand side of a nullary definition then such an occurrence is compiled into  $\text{gcall}_i H$ .

Formally, for any b-list  $\varphi$  and any natural number  $i$

$$(\text{gcall}_i H)_\varphi = H_{\text{link}(i, \varphi, \text{stl}(\varphi))}$$

In the above program then, the occurrence of  $G$  in the definition

$$F(A) = G(A) + X \quad \text{is compiled into} \quad \text{gcall}_0 G$$

Therefore, the first call to  $G$  invoked by the first call of  $F$  is the value of  $G$  at the b-list  $\beta$ , where  $\beta = \text{link}(0, \text{link}(0, \Lambda, \Lambda), \Lambda)$ .

As  $G(C) = C + X$ , the value of  $G$  at the b-list  $\beta$ , is the sum of two values.

The first is the value of  $C$  at  $\beta$ ; and as  $G$  is called once in the whole program then  $C$  is compiled into  $\text{act}(A)$ . By the definition of  $\text{act}$ , described before, the value of  $\text{act}(A)$  at  $\beta$  equals the value of  $A$  at  $\text{dtl}(\beta)$ . As  $A = \text{act}(3)$ , then

the value of  $A$  at  $\text{dtl}(\beta)$  is equivalent to the value of

$3$  at  $\text{dtl}(\text{dtl}(\beta))$  which is the value of  $3$  at  $\Lambda$

which is the value  $3$ .

The second value we have to compute in order to evaluate  $G$  at  $\beta$ , is that of the global  $X$  which is defined in the same clause as  $G$ . We introduce the operator  $\gamma$ , and we translate

$$G(C) = C + X \quad \text{into} \quad G = C + \gamma X$$

Formally, we define the operator  $\gamma$  as follows:

for any b-list  $\varphi$  and for any expression  $\varepsilon$

$$(\gamma \varepsilon)_\varphi = \varepsilon_{\text{stl}(\varphi)}$$

Notice that while  $\text{act}$  switches the context of evaluation (represented by a b-list) to the dynamic tail of the present one,  $\gamma$  switches it to its static tail.

In the example above, the value of  $G$  at  $\beta$  is a function of  $\gamma X$  at  $\beta$ , which is equal to the value of  $X$  at  $\text{stl}(\beta)$  which is  $X$  at  $\Lambda$ .

**Example:**

We give here the full translation of the above example, then we represent

the evaluation process as a tree in which lines represent equalities. The program

```

Z where
  Z = F(3) + X;
  F(A) = G(A) + X;
  G(C) = C + X;
  X = 10;
end

```

is translated into

```

result = Z

Z = call0 F + X

F = gcall0 G + γ X

G = C + γ X

X = 10

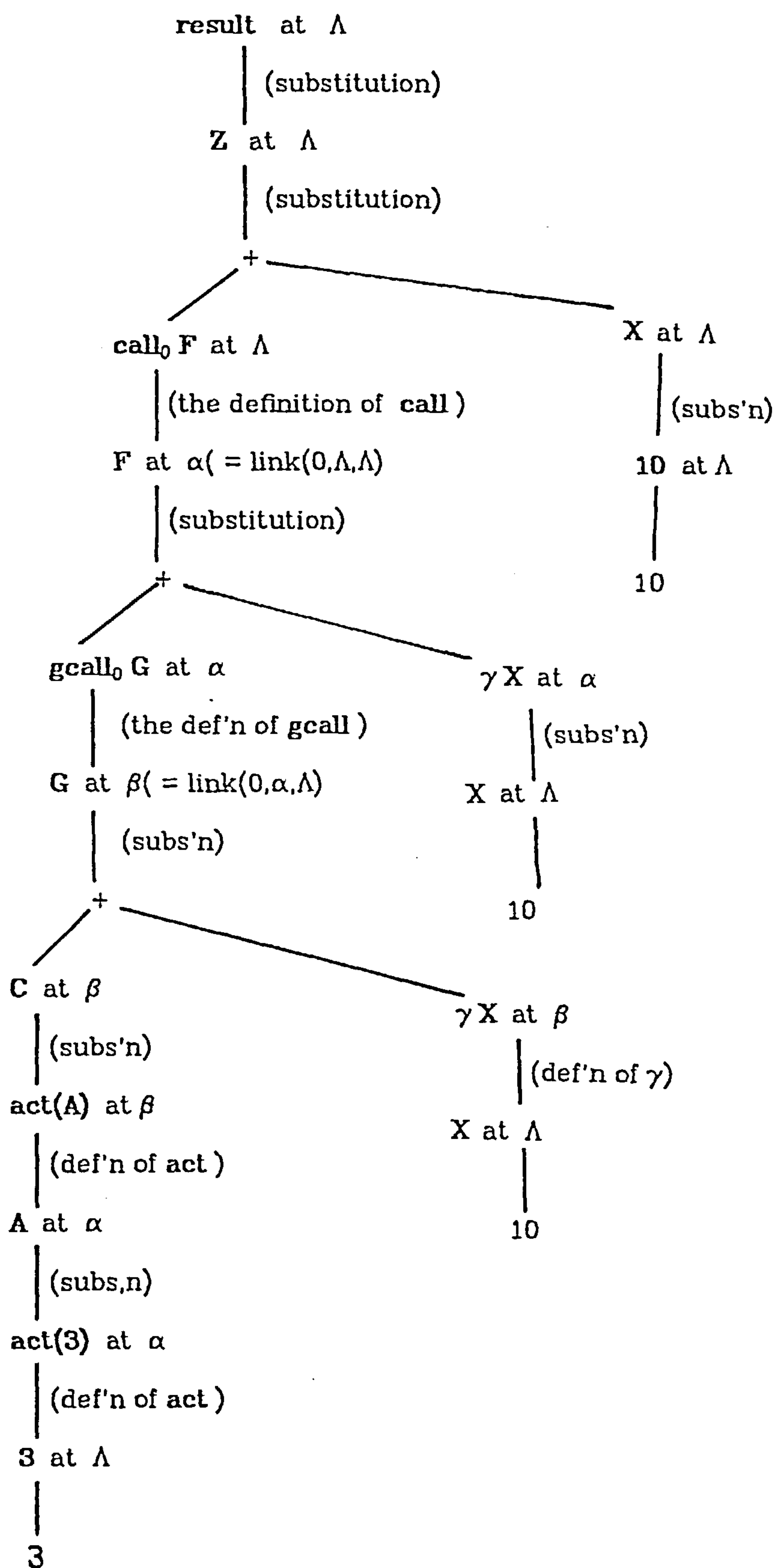
A = act (3)

C = act (A)

```



The value of the program is the value of **result** at  $\Lambda$



#### 4.6.2- Compiling Nullary Globals:

In a structured functional language like Iswum, a global might be imported from a scope level which is well before the present one. For example, in the program

$F(5)$  where

$F(X) = G(X) + A$  where

$G(Y) = A + Y;$

end;

$A = 10;$

end

The occurrence of  $A$  in the definition of  $F$  is global and  $A$  is defined at the same scope level as  $F$ . Therefore, by the discussion in the previous section  $A$  in the definition of  $F$  is compiled into  $\gamma A$ . Moreover, the occurrence of the function symbol  $G$  in the definition of  $F$  is local and it is the first and only occurrence of  $G$  in the program. Thus, again by the discussion above, the definition of  $F$  is compiled into

$F = \text{call}_0 F + \gamma A$

Also,  $F(5)$  is the first (and only occurrence) of  $F$  in the program, and as it is the subject of the main where clause we write

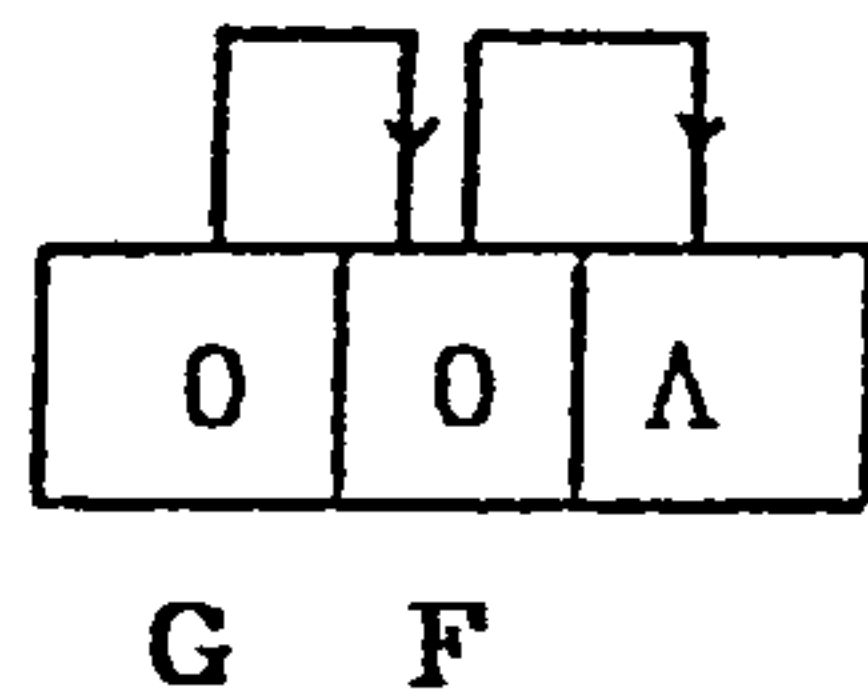
$\text{result} = \text{call}_0 F$

The formal  $X$  of  $F$  is compiled into  $X = \text{act}(5)$

The occurrence of  $A$  in the definition of  $G$  is global and is defined in the outer where clause. It is defined two function definitions outside its occurrence; the definition of  $G$  and that of  $F$ .

Therefore, the first call to  $F$  will invoke the first call to  $G$ ; and hence, the value

of  $G$  is the value of  $A+Y$  at the b-list, say  $\alpha$ , whose head is 0 and dynamic tail equals its static tail which is the b-list  $\langle 0, \Lambda, \Lambda \rangle$ . That is,  $\alpha$  can be illustrated in the diagram



As  $A$  is defined in the outer most where-clause, the value of the occurrence of  $A$  in the definition of  $G$  should be equivalent to the value of  $A$  in the world represented by the empty b-list  $\Lambda$ . That is, it should be evaluated in the world represented by the b-list  $\text{stl}(\text{stl}(\alpha))$

We introduce the new constant symbol  $\gamma\gamma$ , and compile the expression  $A+Y$  defining the symbol  $G$  into

$$\gamma\gamma A + Y; \text{ where for any expression } \varepsilon, \text{ any any b-list } \varphi$$

$$(\text{Flo}(\gamma\gamma\varepsilon))_{\varphi} = (\text{Flo}(\varepsilon))_{\text{stl}(\text{stl}(\varphi))}$$

Similarly, for nullary variable symbols imported through three function definitions, we introduce the constant symbol  $\gamma\gamma\gamma$ , where

$$(\text{Flo}(A)(\gamma\gamma\gamma\varepsilon))_{\varphi} = (\text{Flo}(A)(\varepsilon))_{\text{stl}(\text{stl}(\text{stl}(\varphi)))}$$

Generally then, the signature of  $\text{Flo}$  should contain the set of symbols

$$\{\gamma, \gamma\gamma, \gamma\gamma\gamma, \gamma\gamma\gamma\gamma, \dots, \gamma^n, \dots\}$$

where  $\gamma^n$  for a natural number  $n$  is  $\gamma$  iterated  $n$  times.

For any extensional algebra  $A$ , an expression  $\varepsilon$  and a b-list  $\varphi$ , the value of  $\gamma^n$ , for any natural number  $n$ , is given by

$$(\text{Flo}(A)(\gamma^n \varepsilon))_{\varphi} = (\text{Flo}(A)(\varepsilon))_{\text{stl}^n(\varphi)}$$

where  $\text{stl}^n$  means the application of  $\text{stl}$   $n$  times.

**Example:** A formal compilation algorithm will be given in section 4.3. However,

we give here the compilation of the above discussed Iswum-program according to our informal analysis

```

F(5) where
  F(X) = G(X) + A where
    G(Y) = A + Y;
  end;
  A = 10;
end

```

is compiled into

```
result = call0 F;
```

```
F = call0 G + γA;
```

```
G = γγ A + Y;
```

```
A = 10 ;
```

```
X = act (5) ;
```

```
Y = act (X) ;
```

#### 4.6.3- Compiling Global Functions:

Consider the following program in Iswum:

```

F(3) where
  F(A) = A + G(X) where
    G(B) = H(B*2) + X;
    X = 4 + H (A) ;
  end;
  H(C) = C + 2;
end

```

The function symbol  $H$  is defined in the outer most where-clause. However, it occurs as a global in both the definition of  $X$  and that of  $G$  in the inner where-clause. Thus, any call to  $F$  will produce eventually two calls to  $H$ . The first is invoked through the call to  $G$  and the other through the evaluation of the nullary symbol  $X$ .

As the subject of the main where clause is the first occurrence of  $F$ , then

```
result = call0 F
```



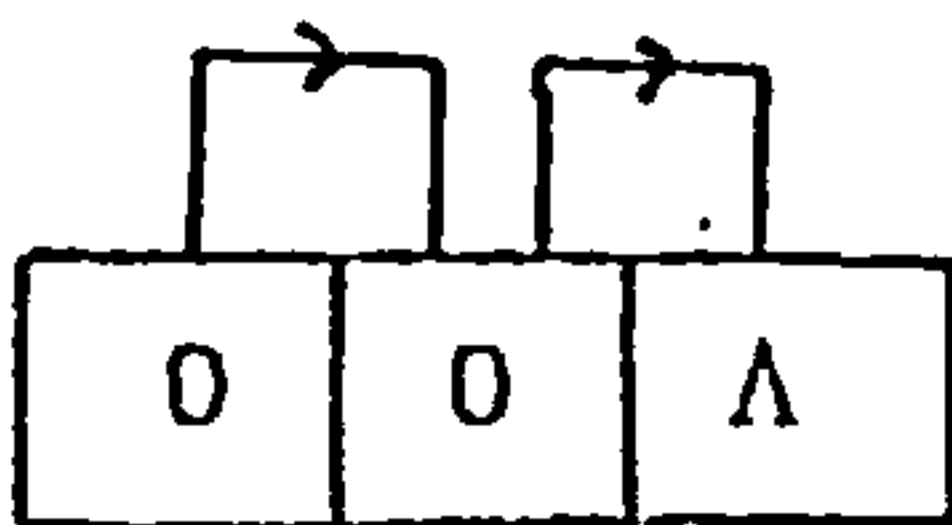
The value of the program above, then, is the value of  $F$  at the b-list  $\alpha$  where  $\alpha$  is  $\langle 0, \Lambda, \Lambda \rangle$ .

Also, the occurrence of  $G$  in the definition of  $F$  is local. Therefore,

$$F = A + \text{call}_0 G$$

Hence by substitution, the value of  $F$  at  $\alpha$  is equivalent to the value of  $A + \text{call}_0 G$  at  $\alpha$ .

By the definition of  $\text{call}$  then, the value of  $F$  is a function of the value of  $G$  at the b-list  $\beta$  where  $\beta = \text{link}(0, \alpha, \alpha)$ . That is,  $\beta$  is the b-list illustrated as

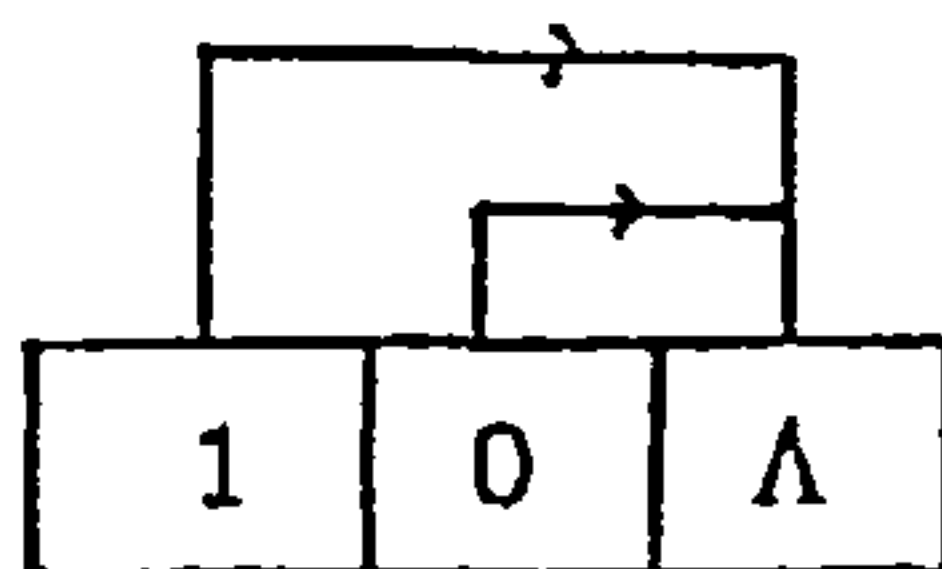


the b-list  $\beta$

As the occurrence of  $X$  in the definition of  $G$  is global, then by the discussion in section 4.6.1,  $X$  should be compiled into  $\gamma X$ . Therefore, the value of  $G$  at  $\beta$  will be a function of the value  $\gamma X$  at  $\beta$ , which is equivalent to the value of  $X$  at  $\text{stl}(\beta)$ . That is, it is the value of  $X$  at the b-list  $\alpha$ . Notice that  $H$  is defined in the main where clause (at the same scope level as the definition of  $F$ ), but its second occurrence (in the definition of  $X$ ) is within the definition of  $F$ . Therefore, it is compiled into  $\text{gcall}_1 H$ . That is,

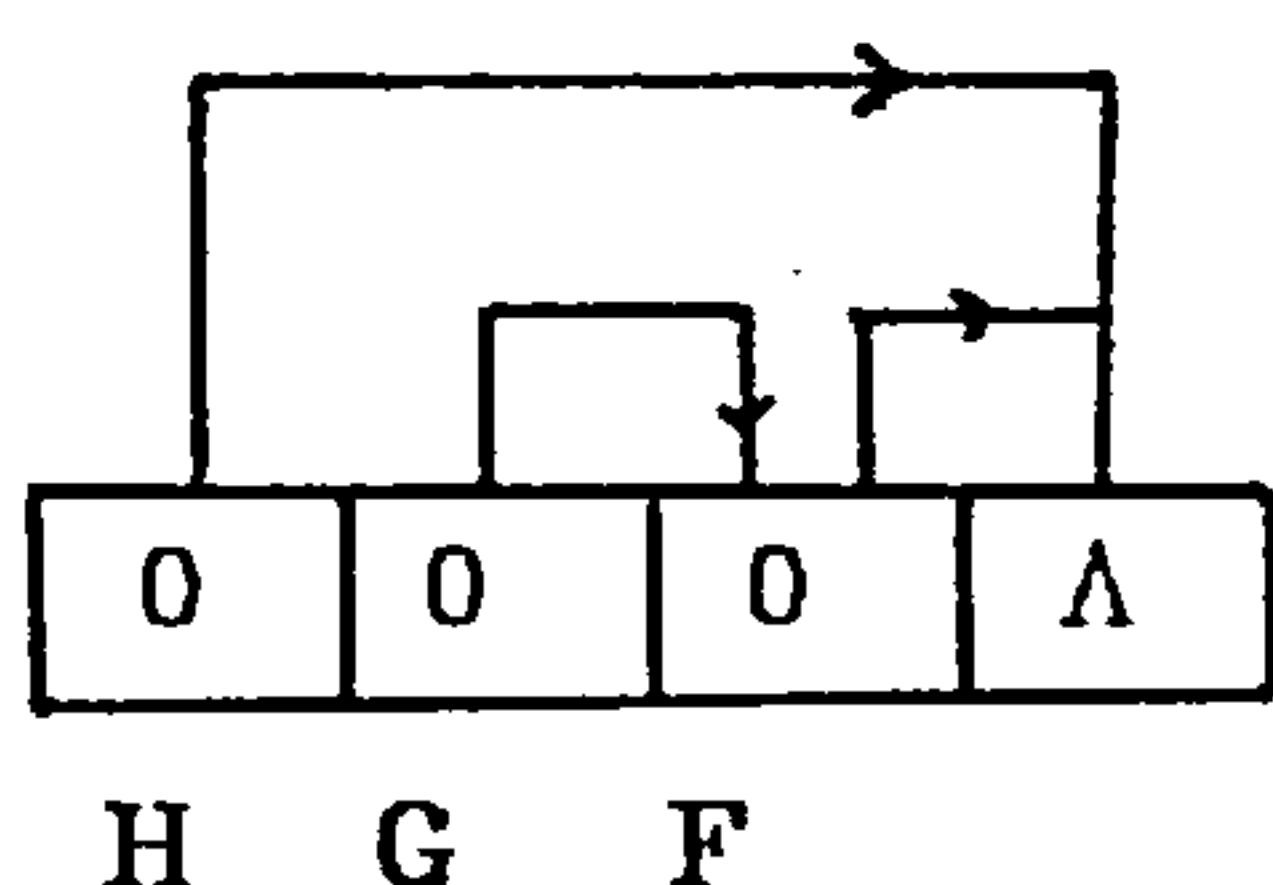
$$X = 4 + H(A) \text{ is compiled into } X = 4 + \text{gcall}_1 H$$

Consequently, by the definition of  $\text{gcall}$ , the value of  $X$  at  $\alpha$  is a function of the value of  $H$  at the b-list  $\vartheta$  where  $\vartheta$  is illustrated as



the b-list  $\vartheta$

However, when the call to  $G$  invokes the call to  $H$ , the latter should be evaluated at the b-list whose head is  $0$  (because the occurrence of  $H$  in the definition of  $G$  is the first occurrence in the program) and whose dynamic tail is  $\beta$ . The static tail of such a b-list should be  $\Lambda$  because  $H$  is defined at the outer most where-clause. That is, the first call to  $H$  should be evaluated at the b-list illustrated as



It is worth noting here that the static tail of the b-list above is the static tail of the static tail of its dynamic tail. This is because the dynamic tail is a reference to the occurrence of the function symbol (or semantically, the call to the function), while the static tail is a reference to the definition of the function. In the source program, the function  $H$  is two function definitions (levels) outside its occurrence level.

For such a case, we define the family of intensional operators  $ggcall$ .

For a b-list  $\varphi$ , an expression  $\varepsilon$ , and  $i \in \omega$

$$(ggcall_i \varepsilon)_{\varphi} = \varepsilon_{\text{link}(i, \varphi, \text{st}(\text{st}(\varphi)))}$$

Thus the program

```

F(3) where
  F(A) = A + G(X) where
    G(B) = H(B*2) + X;
    X = 4 + H(A);
  end;
  H(C) = C + 2;
end

```

is translated into the DE-program

```

result = call0 F
F = A + call0 G
G = ggcall0 H + γ X
X = 4 + gcall1 H
H = C + 2
A = act (3)
B = act (X)
C = act (B*2 , A)

```

Similarly, we introduce the family of operators *gggcall*. If the *i*'th occurrence of a function symbol *Q* is three function definitions deeper than its definition, then such an occurrence is compiled into *gggcall<sub>i</sub> Q*.

Formally speaking, for every  $i \in \omega$ , every expression  $\varepsilon$ , and every b-list  $\varphi$

$$(gggcall_i \varepsilon)_\varphi = \varepsilon_{\text{link}(i, \varphi, \text{sl}(\text{sl}(\text{sl}(\varphi))))}$$

and so forth with the families *ggggcall*, *ggggggcall* ...etc.

Therefore, the signature of *Flo* should contain the families of symbols

$$\{\text{call}_i : i \in \omega\}, \{\text{gcall}_i : i \in \omega\}, \{\text{ggcall}_i : i \in \omega\}, \{\text{gggcall}_i : i \in \omega\} \dots \text{etc.}$$

In other words, it should contain the family of operator symbols

$$\{g^n \text{call}_i : i, n \in \omega\}$$

where  $g^n \text{call}$  for any natural number  $n$  is

the symbol  $g \cdots g \text{call}$  where  $g$  is

iterated  $n$  times.

In general, for every  $i, n \in \omega$ ,

every expression  $\varepsilon$

and every b-list  $\varphi$

$$(g^n \text{call}_i \varepsilon)_\varphi = \varepsilon_{\text{link}(i, \varphi, \text{stl}^n(\varphi))}$$

where  $\text{stl}^n$  denotes the application of  $\text{stl}$   $n$  times.

It is worth pointing out here that we do not consider the letter  $g$  in  $g\text{call}$  as an operator on  $\text{call}$ . That is,  $g\text{call}$  is not  $g(\text{call})$  but a family of indexed operator symbols; it is the family  $\{g\text{call}_i : i \in \omega\}$ . So are the symbols  $\text{call}$ ,  $gg\text{call}$  and  $ggg\text{call}$ .

Also, the same should be said for the operator symbols generated by  $\gamma^n$ . Thus  $\gamma\gamma$  is not  $\gamma(\gamma)$



#### 4.7- The Definition of the Algebra $Flo$ :

$Flo$  is, like  $FUN$  and  $Lat$ , a function which maps extensional algebras to intensional ones. It defines the family of intensional algebras

$$\{Flo(A) : A \text{ is an extensional algebra}\}$$

In this section we give the formal definition of  $Flo$ .

Given an extensional  $\Sigma$ -algebra  $A$ ,  $Flo(A)$  is the intensional  $\Sigma$ -algebra whose

universe  $U$  is the set  $bl(\omega)$  (the set of lists with back pointers over the natural numbers)

and whose signature is the union

$$\Sigma \cup \{act\} \cup \{g^n call_i : i, n \in \omega\} \cup \{\gamma^n : n \in \omega\}$$

such that

$Flo(A)$  extends  $A$  point wise. That is,

for every  $n$ -ary constant symbol  $\psi$  in  $\Sigma$ ,

any sequence of  $Flo(A)$ -expressions  $x_0, \dots, x_{n-1}$ ,

and any  $u \in U$

$$(Flo(A)(\psi)(x_0, \dots, x_{n-1}))_u = A(\psi)(Flo(A)(x_0)_u, \dots, Flo(A)(x_{n-1})_u)$$

For every  $u \in U$ , and every sequence of expressions  $x_0, \dots, x_{n-1}$

$$(Flo(A)(act(x_0, \dots, x_{n-1})))_u = Flo(A)(x_{hd(u)})_{tl(u)}$$

For every expression  $\varepsilon$ , and every  $u \in U$

$$(Flo(A)(\gamma^n \varepsilon))_u = Flo(A)(\varepsilon)_{stl^n(u)}$$

where  $\gamma^n$  is the string of catenating  $\gamma$   $n$  times.

and  $stl^n$  is the application of the function  $stl$   $n$  times.

That is,

$$(Flo(A)(\gamma \varepsilon))_u = Flo(A)(\varepsilon)_{stl(u)}$$

$$(Flo(A)(\gamma\gamma\epsilon))_u = Flo(A)(\epsilon)_{stl(stl(u))}$$

$$(Flo(A)(\gamma\gamma\gamma\epsilon))_u = Flo(A)(\epsilon)_{stl(stl(stl(u)))}$$

and so on for any  $n \in \omega$ .

For every expression  $\epsilon$ ,

every  $i, n \in \omega$

and every  $u \in U$

$$(Flo(A)(g^n call_i \epsilon))_u = Flo(A)(\epsilon)_{link(i, u, stl^n(u))}$$

That is, for any  $Flo(A)$ -expression  $\epsilon$ ,

for every  $i \in \omega$ ,

and for every  $u \in U$

$$(Flo(A)(call_i \epsilon))_u = Flo(A)(\epsilon)_{link(i, u, u)}$$

$$(Flo(A)(gcall_i \epsilon))_u = Flo(A)(\epsilon)_{link(i, u, stl(u))}$$

$$(Flo(A)(ggcall_i \epsilon))_u = Flo(A)(\epsilon)_{link(i, u, stl(stl(u)))}$$

$$(Flo(A)(gggcall_i \epsilon))_u = Flo(A)(\epsilon)_{link(i, u, stl(stl(stl(u))))}$$

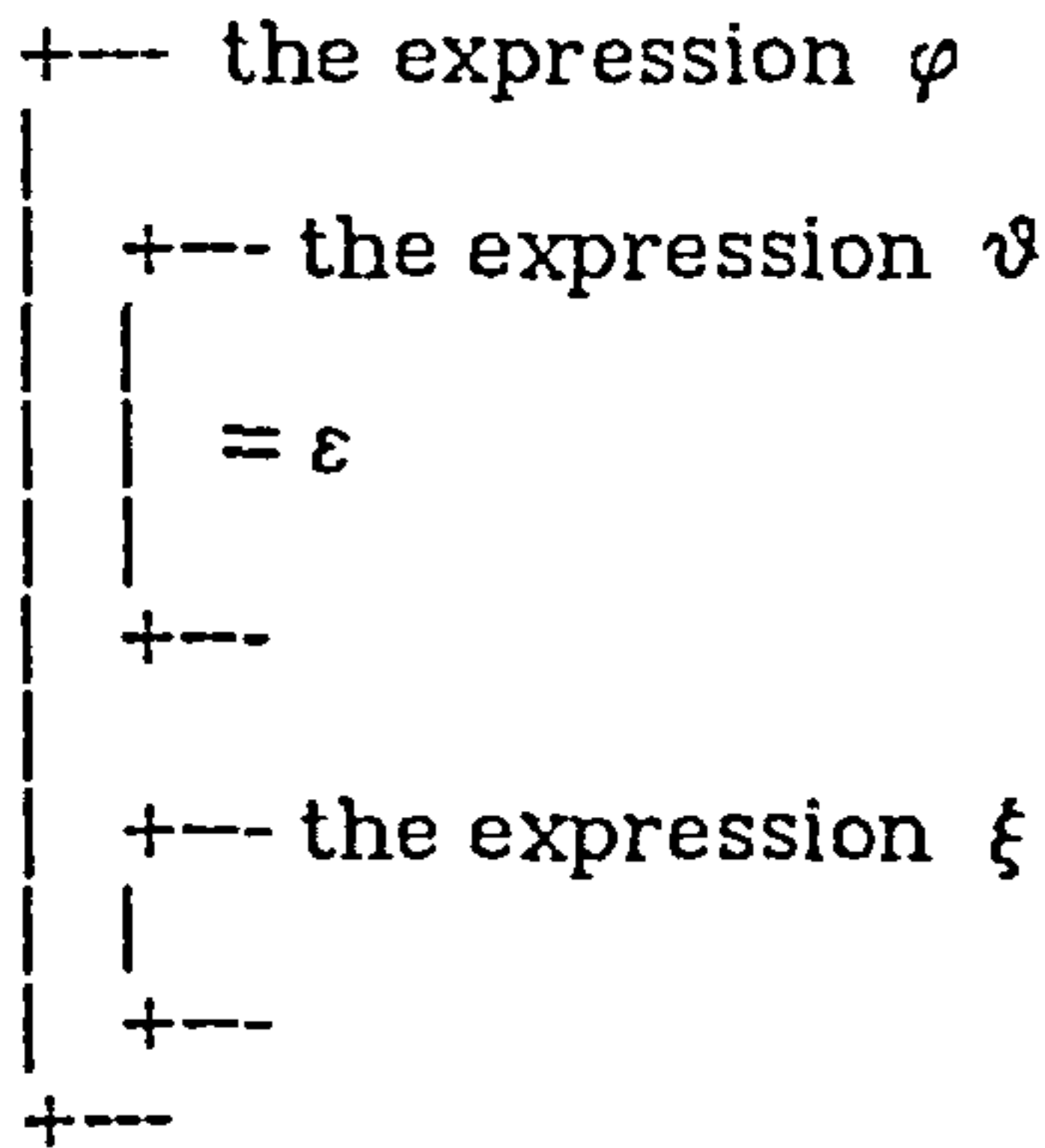
and so on for any  $n \in \omega$ .

#### 4.8- The Compilation of Iswum into Florid:

While a program in Iswum is a valid expression, a program in Florid is a set of equations over intensional expressions. The translation algorithm, then, will be a function mapping each Iswum-program into a set of Florid-equations. The compatibility required in the target Florid-program is easily captured due to the compatibility of the source Iswum-program, which is captured by the renaming rules described before.

The important fact we want to stress here is that translation (compilation) will never be done in a vacuum. When we compile an expression, we have to compile it relative to the where-clause it appears in. This is so that we can determine whether the occurrences of variable symbols in the expression are locals or globals. Moreover, we need to know all the where-clause expressions which contain the present one so that we bind the variable symbols in the expression to their definitions properly.

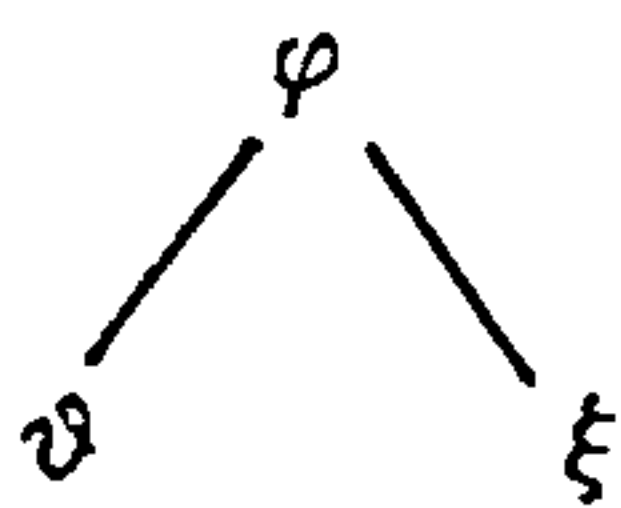
Therefore, we shall talk about the compilation of an expression  $\varepsilon$  relative to a list of expressions. The head of such a list is the where-clause containing  $\varepsilon$ , and the tail is the sequence of expressions containing the present where-clause. For example, suppose we have an Iswum program whose structure is represented by the diagram



The expression  $\varepsilon$  then should be compiled relative to the list of expressions

$[\vartheta, \varphi]$ . The expression  $\varphi$ , however, should be compiled relative to the empty list  $[]$ . We shall denote the compilation of the expression  $\varepsilon$  above by  $\text{compexp}_{[\vartheta, \varphi]}(\varepsilon)$ , and that of  $\vartheta$  by  $\text{compexp}_{[\varphi]}(\vartheta)$ .

We can view a structured program in Iswum as a tree whose nodes are where-clauses and whose edges represent the containment relation between these clauses. That is, a where-clause  $\vartheta$  is contained in the clause  $\varphi$  if  $\vartheta$  occurs in the right hand side of a definition in the body of  $\varphi$ . For example, the above structure can be represented by the tree



The compilation procedure then will flatten a structured Iswum-program by traversing the tree of where-clauses of the program.

In the algorithm, we use the symbol  $\wedge$  to denote the concatenation operator on strings. For example, if **xyz** and **ab** are two strings then

$$\mathbf{xyz} \wedge \mathbf{ab} = \mathbf{xyzab}.$$

Moreover, given a where-clause expression  $\varphi$ , we say that a variable symbol  $v$  is global in  $\varphi$ , denoted by  $\text{global}_{\varphi}(v)$  if  $v$  is neither the left hand side of any equation in the body of  $\varphi$  nor it is a formal parameter of  $\varphi$ . Otherwise, we say  $\text{local}_{\varphi}(v)$ .

In the compilation of function application however, our interest is mainly with a particular occurrence at a time; that is, a particular function call. Therefore, in the algorithm we talk about the  $i$ 'th occurrence of a certain function symbol, say  $F$ , and denote that by  $(i, F)$ .



#### 4.8.1- The Compilation Algorithm:

Given a program  $P$  in  $\text{Iswum}(A)$ , for an extensional algebra  $A$ , the target program  $\text{Trans}(P)$  in  $\text{Florid}(A)$ , where  $\text{Florid}(A)$  is  $DE(\text{Flo}(A))$ , is defined recursively as follows:

if  $P$  is a simple expression in  $\text{Iswum}(A)$  then  $\text{Trans}(P)$  is the singleton

$$\{ \text{result} = \text{compexp}_{[P]} P \}$$

if  $P$  is of the form

$$\delta \text{ where } D \text{ end}$$

where  $\delta$  is a simple expression and  $D$  is a set of  $\text{Iswum}(A)$ -equations

then  $\text{Trans}(P)$  is

$$\{ \text{result} = \text{compexp}_{[P]}(\delta) \} \cup \{ \text{compdef}_{[P]}(d) \mid d \in D \}$$

where for a list of expressions  $\Phi$ , and for the expression  $\varepsilon$

$$\text{compexp}_{\Phi}(\varepsilon) =$$

if  $\varepsilon$  is of the form  $\psi(x_0, \dots, x_{n-1})$

where  $\psi$  is an  $n$ -ary constant symbol and

$x_0, \dots, x_{n-1}$  are  $n$  expressions,

then

$$\psi(\text{compexp}_{\Phi}(x_0), \dots, \text{compexp}_{\Phi}(x_{n-1}))$$

else if  $\varepsilon$  is a nullary variable symbol then  $\text{compvar}_{\Phi}(\varepsilon)$

else if  $\varepsilon$  is the  $i$ 'th occurrence of the  $n$ -ary function symbol

$F$  and  $x_0, \dots, x_{n-1}$  are the actuals of such

an occurrence then

$$\text{compfun}_{\Phi}(i, F)$$

$\text{compvar}_{\Phi}(v) =$

if  $\text{global}_{\text{hd}(\Phi)}(v)$  and  $\text{tl}(\Phi) \neq []$   
     then  $\gamma^{\wedge}(\text{compvar}_{\text{tl}(\Phi)}(v))$   
     else  $v$

$\text{compfun}_{\Phi}(i, F) =$

if  $\text{tl}(\Phi) \neq []$  and  $\text{global}_{\text{hd}(\Phi)}(i, F)$   
     then  $g^{\wedge}(\text{compfun}_{\text{tl}(\Phi)}(i, F))$   
 else if  $\text{local}_{\text{hd}(\Phi)}(i, F)$  then  $\text{call}_i F$   
     else  $\perp$

$\text{compdef}_{\Phi}(d) =$

if  $d$  is of the form  $V = \varepsilon$ , where  $V$  is a nullary  
     variable symbol, and  $\varepsilon$  is a simple expression  
     in  $\text{lswum}(A)$ , then  
      $\{V = \text{compexp}_{\Phi}(\varepsilon)\}$   
 else if  $d$  is of the form  $F(x_0, \dots, x_{n-1}) = \varepsilon$   
     where  $\varepsilon$  is a simple expression in  $\text{lswum}(A)$ ,  
      $F$  is an  $n$ -ary variable symbol, and  $x_0, \dots, x_{n-1}$   
     is the list of formal parameters, then  
      $\{F = \text{complexp}_{\Phi} \varepsilon\} \cup \{\text{compform}_{\text{hd}(\Phi), F}(x_i) \mid i \in n\}$   
 else if  $d$  is of the form  $F(x_0, \dots, x_{n-1}) = \varepsilon$   
     where  $\varepsilon$  is a where-clause expression of the form  
      $\alpha$  where  $D$  end  
     then  
      $\{F = \text{compexp}_{\text{cons}(\varepsilon, \Phi)} \alpha\}$   
      $\cup \{\text{compform}_{\text{hd}(\Phi), F}(x_i) \mid i \in n\}$   
      $\cup \{\text{compdef}_{\text{cons}(\varepsilon, \Phi)}(d) : d \in D\}$

$\text{compglexp}_{\Phi}(\varepsilon) =$

if  $\varepsilon$  is of the form  $\psi(x_0, \dots, x_{n-1})$

where  $\psi$  is an  $n$ -ary constant symbol and

$x_0, \dots, x_{n-1}$  are  $n$  expressions, then

$\psi(\text{compglexp}_{\Phi}(x_0), \dots, \text{compglexp}_{\Phi}(x_{n-1}))$

else if  $\varepsilon$  is a formal then  $\varepsilon$

else if  $\varepsilon$  is a nullary variable symbol then

$\gamma^{\sim} \text{compvar}_{\Phi}(\varepsilon)$

else if  $\varepsilon$  is the  $i$ 'th occurrence of the  $n$ -ary function  $F$

together with  $x_0, \dots, x_{n-1}$  as the list of

actuals of such an occurrence, then

$g^{\sim}(\text{compfun}_{\Phi}(i, F))$

$\text{compform}_{\Phi, F}(x) = \{x = \text{act}(\text{compexp}_{\Phi}(a_0), \dots, \text{compexp}_{\Phi}(a_{m-1}))\}$

where for each  $i$  in  $m$ ,  $a_i$  is the actual of

the  $i$ 'th occurrence of  $F$  in  $\Phi$ .

#### 4.8.2- Remarks on the algorithm:

Notice in the algorithm that , the  $i$ 'th occurrence of the function symbol,  $F$  say, is translated into either  $\text{call}_i F$ , or  $g^n \text{call}_i F$  depending on whether such an occurrence is local or global. Each actual of such an occurrence, however, will be the  $i$ 'th element in the sequence of expressions defining the corresponding formal. The main point we have to ensure is the correspondence between function calls and lists of actuals. Thus if an occurrence of a function is considered to be the  $i$ 'th occurrence then the actual of such an occurrence (assuming for simplicity that the function is nullary) should be the  $i$ 'th element in the list of actuals defining the formal of the function.

Indexing occurrences of functions by counting from left to right or right to left, is dependent on the implementation and the representation of the source language. For example, in [Wad84] and [Yag84a] a program in the source language is represented as a structured list. Thus counting occurrences of function calls can be done recursively by traversing the list.



## 4.8.3- Example:

The following Iswum-program

$F(x,y)$  where

$F(a,b) = a*a + G(2*c)$  where

$G(d) = d + H(d) + v$ ;

$c = H(a) + G(b) + v$ ;

end;

$v = 10$ ;

$H(w) = w*w*w$ ;

end

is translated into:

result = call<sub>0</sub> F;

$F = a * a + \text{call}_0 G$ ;

$G = d + \text{ggcall}_0 H + \gamma\gamma v$ ;

$c = \text{gcall}_1 H + \text{call}_1 G + \gamma v$ ;

$v = 10$ ;

$H = w*w*w$ ;

$a = \text{act } (x)$ ;

$b = \text{act } (y)$ ;

$d = \text{act } (2*c, b)$ ;

$w = \text{act } (d, a)$ ;

# Chapter 5

## Rewrite Rules and Evaluation Techniques for Florid

### 5.0- Introduction:

We have described the process of compiling the family of languages Iswim into the target language Florid. The language Florid has two main properties:

(1) It is an equational single assignment language; i.e. a program is a set of compatible equations. Compatible means that a variable symbol is defined at most once in the whole program. This property makes Florid an appropriate code for dataflow machines.

(2) It is an intensional language; i.e. the value of a Florid-expression changes from one context to another and Florid-expressions contain intensional operators such as `call`, `gcall`. This property makes Florid a suitable code for tagged data flow machines. It is suitable since the tagging mechanism in the evaluation of Florid expressions is based on a mathematical model which is the intensional algebra *Flo*. Therefore, offers a wide range of evaluation techniques for functional languages whether on conventional machines or on parallel ones such as dataflow machines.

In this chapter we describe two techniques for evaluating Florid-programs. The first is **reductive** and is based on a set of rewrite rules for the algebra *Flo*, the second is **eductive** and is based on demand driven tagged data flow architecture or what has been termed *education*.

Our main interest in this thesis is exclusively with the eduction approach to evaluating intensional expressions. The reduction method is probably impractical; it is presented here, however, mainly to emphasize the difference between eduction and reduction.

### 5.1- Rewrite Rules and Reduction Sequences:

Equations over algebraic terms, term rewriting systems and reduction sequences are the subject of active research in computer science. For more details on this field, the reader is referred to [HoOd82, Odo77, HuOp80, Mus78, HuLe79].

Given a  $\Sigma$ -algebra  $A$ , an equation over the algebra  $A$  is of the form

$$L = R$$

where  $L$  and  $R$  are expressions formed from the operation symbols of the algebra  $A$  together with variables ranging over the elements of the domain of  $A$ .

We say that an equation  $L = R$  is valid in an algebra  $A$ , or equivalently an algebra  $A$  satisfies the equation  $L = R$ , if and only if the value of  $L$  in any  $A$ -environment is equivalent to the value of  $R$  in such an environment. That is, the two sides of the equation have the same value in the domain of  $A$  irrespective of how the variable symbols are interpreted. We say that an algebra  $A$  satisfies a set of equations if the algebra simultaneously satisfies every equation in the set.

For example, given the algebra  $Z$  of the integers, represented by their usual mathematical symbols, the following equations are valid

$$X+Y = Y+X \quad (\text{the commutativity property})$$

$$X+(Y+Z) = (X+Y)+Z \quad (\text{the associativity property})$$

$$X+0 = X \quad (\text{the identity element})$$

$$X + (-X) = 0 \quad (\text{the inverse})$$

Thus given any  $Z$ -environment  $\varepsilon$  (an environment which assigns integers to variable symbols), the value of  $X+Y$  in  $\varepsilon$  is equivalent to the value of  $Y+X$  in  $\varepsilon$ .

Equations over algebras of data types, such as the above set, state properties which hold between the objects of the type. Hence, they are efficient tools in the specification of, and the reasoning about, abstract data types. Our interest here, however, is exclusively with the use of such equations in evaluating equational languages or with *term rewriting rules*.

Rewrite rules are a special type of equations; they are **directed equations**. The difference between arbitrary equations over a certain algebra and the rewrite rules of such an algebra is that the equality relation in the former is symmetric; that is

$$C = B \text{ implies } B = C$$

while in rewrite rules the equality is directional or the equality relation is one sided. Of course, rewrite rules are still equations and are satisfied by the algebra; that is if  $C=B$  is a rewrite rule then  $C$  equals  $B$  and  $B$  equals  $C$ . Their name (rewrite rules) is just a pragmatic one, being rules for purely syntactic rewriting of the expressions of the algebra.

That is, given the algebra  $A$ , the rewrite rule

$$C = B$$

of  $A$  states the fact that  $B$  can be substituted for any occurrence of  $C$  in any expression in the algebra  $A$ . Informally speaking, we can say that the right hand side of a rewrite rule is always computationally simpler than its right hand side. Thus, rewriting any occurrence of the left hand side by the right hand side is a step towards simplifying the expression and hence evaluating it. Such a step is called a **reduction step**.



Evaluation by reduction is a sequence of reduction steps; that is, it is repeatedly replacing occurrences of left hand sides by the corresponding right hand sides until reaching an expression with no occurrences of left hand sides. The sequence of expressions which results from this process is called a **reduction sequence**. Thus, a reduction sequence is a sequence of equivalent expressions  $e_0, \dots, e_{n-1}$  where  $e_0$  is the expression to be reduced or computed and, for any  $i < n$ ,  $e_i$  is obtained from  $e_{i-1}$  by purely syntactic substitution of an occurrence of a left hand side of a rewrite rule by the right hand side of the rule.

The target language *DE* is a family of equational single assignment languages (see 2.8). A program in *DE*(A), for an algebra A, is a compatible set of nullary equations over the algebra A. One of these equations defines the variable symbol **result**. In *DE*, evaluation by reduction is a straightforward process. Given a program P in *DE*(A), by the semantics of *DE*, the first expression in the reduction sequence should be the variable symbol **result**. Any reduction step thereafter should be a rewriting of an occurrence of a certain subexpression according to either an equation in the program or a rewrite rule for the algebra A.

Consider, for example, the following *DE*(Q)-program, where Q is the algebra of the rationals, represented by their usual symbols.

**result** =  $X + Y - Z / X$ ;

$Y = 3 * X - 5$ ;

$X = 10 * Z$ ;

One possible reduction sequence for this program might be:

**result** ,  $X + Y - Z / X$  ,  $(10 * Z) + Y - Z / (10 * Z)$  ,

$(10 * Z) + (3 * X - 5) - Z / (10 * Z)$  ,  $(10 * Z) + (3 * (10 * Z) - 5) - Z / (10 * Z)$  ,

$(10 * Z) + (30 * Z) - 5 - 1 / 10$  ,  $40 * Z - 5 - 1 / 10$  ,  $40 * Z - 49 / 10$

Notice that the reduction step from

$$(10 * Z) + (3 * (10 * Z) - 5) - Z / (10 * Z) \quad \text{to} \quad (10 * Z) + (30 * Z) - 5 - 1 / 10$$

is a consequence of the *Q*-rules

$$A * 1/A = 1 \quad (\text{the inverse property})$$

$$A * (B * C) = (A * B) * C \quad (\text{the associativity property})$$

The reader should notice here that the rewrite rules of an algebra *A* are reduction rules for evaluating any *DE*(*A*)-program, while the equations of a *DE*(*A*)-program are used as reduction rules for evaluating only the program itself.

We give here the rewrite rules of the algebra *Flo* together with a complete justification of their correctness using the definition of *Flo*.

## 5.2- Rewrite Rules for the Algebra *Flo*

*Flo* is a family of intensional algebras; each member *Flo*(*A*) is uniquely determined by an algebra of data types *A*. The rewrite rules of any member *Flo*(*A*), then, are also determined by the rewrite rules of *A*. What we will introduce here is the set of rewrite rules for the whole family *Flo* irrespective of *A*. The only rule which concerns the algebra *A* is the first one. It states that the operators of *Flo* are distributive over those of *A*. For example, in *Flo*(*Q*), where *Q* is the extensional algebra over the rationals, the following equation holds

$$\text{call}_i (X + Y - 3) = \text{call}_i X + \text{call}_i Y - \text{call}_i 3$$

The rewrite rules of *Flo*(*A*), for an algebra *A*, are equations over the terms of *Flo*(*A*). This means that both sides of each rule are *Flo*(*A*)-terms. Thus, to justify a certain rule is to show that both sides of the rule denote the same object (have the same meaning). Moreover, as *Flo*(*A*) is intensional, the value of

a term is a family of objects each of which denotes its value at a world in the universe. Thus, to show the correctness of a rule, we have to show that the equality holds for all the worlds in the universe.

For example, to justify the rewrite rule  $A=B$ , for the algebra  $\mathbf{Flo}(A)$  with universe  $U$ , we have to show that

for every world  $u \in U$

$$(\mathbf{Flo}(A)(A))_u = (\mathbf{Flo}(A)(B))_u$$

The reader should note that each of the equations given below produces a family of rewrite rules. Each rule in such a family is concerned with a particular family of symbols generated by a particular  $n$  in  $\omega$ . For example, the symbol  $g^n\text{call}$  in  $\mathbf{Flo}$  generates a family of operator symbols. It generates the family

$$\{\text{call}, g\text{call}, gg\text{call}, ggg\text{call}, \dots, g^n\text{call}, \dots : n \in \omega\}$$

Each member of this family is also a family of operator symbols;

$\text{call}$  is the family of symbols  $\{\text{call}_i : i \in \omega\}$

$g\text{call}$  is the family of symbols  $\{g\text{call}_i : i \in \omega\}$

$gg\text{call}$  is the family of symbols  $\{gg\text{call}_i : i \in \omega\}$

and so on.

Thus rule 1, given below, and states that

$$g^n\text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i \text{ for any } n \in \omega,$$

produces a family of directed equations (rewrite rules). Each of these rules is for a particular family of symbols generated by a particular  $n \in \omega$ . For example,

for the family  $\{\text{call}_i : i \in \omega\}$  (i.e. when  $n=0$ ), the rule generates

$$\text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i$$

for the family  $\{g\text{call}_i : i \in \omega\}$  (i.e. when  $n=1$ ), it generates

$$g\text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i$$

and so on.



### 5.2.1- The Rules of *Flo*:

Let  $A$  be an extensional  $\Sigma$ -algebra, then

(Rule 0) If  $\psi$  is an  $n$ -ary operator symbol in the signature of  $A$ ,

and  $x_0, \dots, x_{n-1}$  are  $Flo(A)$ -expressions, then

for every operator symbol  $\vartheta$  in the signature of  $Flo$

$$\vartheta(\psi(x_0, \dots, x_{n-1})) = \psi(\vartheta(x_0), \dots, \vartheta(x_{n-1}))$$

If  $X, X_0, \dots, X_{m-1}$  are  $Flo(A)$ -terms, then

(Rule 1)  $g^n \text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i$  for any  $n \in \omega$

This rule produces a family of directed equations (rewrite rules). Each of these rules is for a particular family of symbols generated by a particular  $n \in \omega$ .

The rules are:

(Rule 1.1)  $\text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i$

for the family  $\{\text{call}_i : i \in \omega\}$

(Rule 1.2)  $g\text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i$

for the family  $\{g\text{call}_i : i \in \omega\}$

(Rule 1.3)  $gg\text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i$

for the family  $\{gg\text{call}_i : i \in \omega\}$

and so on.

(Rule 2) For any  $n \in \omega$ ,  $g^n \text{call}_i(\gamma X) = \gamma^n X$

This rule also generates a family of rules, each corresponds to a particular  $n$  in  $\omega$ . The rules are

(Rule 2.1)  $\text{call}_i(\gamma X) = X$  for the family  $\{\text{call}_i : i \in \omega\}$

(Rule 2.2)  $g\text{call}_i(\gamma X) = \gamma X$  for the family  $\{g\text{call}_i : i \in \omega\}$

(Rule 2.3)  $gg\text{call}_i(\gamma X) = \gamma\gamma X$  for the family  $\{gg\text{call}_i : i \in \omega\}$

(Rule 2.4)  $ggg\text{call}_i(\gamma X) = \gamma\gamma\gamma X$  for the family  $\{ggg\text{call}_i : i \in \omega\}$

and so on.

**Proof:**

Let  $A$  be an extensional algebra.

Let  $U$  be the universe of  $\mathbf{Flo}(A)$  (i.e.  $U = \text{bl}(\omega)$ ).

For simplicity, we shall denote  $(\mathbf{Flo}(A)(x))_u$ , for any  $\mathbf{Flo}(A)$ -expression  $x$ , by  $[x]_u$  where  $u \in U$ .

For the  $\mathbf{Flo}(A)$ -terms  $x, x_0, \dots, x_{m-1}$ ,

for every  $u \in U$ , and for every  $i \in \omega$ .

(Rule 0) Proving (Rule 0) is straightforward and a direct consequence of the fact that the operator symbols of the algebra  $A$  are interpreted pointwise. Notice that this rule also produces a family of directed equations each of which states the distributive property for a certain operator in  $\mathbf{Flo}$ . We prove here the case  $\gamma^n$  only; i.e. we prove that

for any operator symbol  $\psi$  in  $\Sigma$ , and any  $m$  expressions

$x_0, \dots, x_{m-1}$

$\gamma^n(\psi(x_0, \dots, x_{m-1})) = \psi(\gamma^n(x_0), \dots, \gamma^n(x_{m-1}))$  for any  $n \in \omega$ .

Thus, we have to prove that for any  $u \in U$ ,

$$[\gamma^n(\psi(x_0, \dots, x_{m-1}))]_u = [\psi(\gamma^n(x_0), \dots, \gamma^n(x_{m-1}))]_u$$

Let  $n \in \omega$  and  $u \in U$ , then

$$\begin{aligned} & [\gamma^n(\psi(x_0, \dots, x_{m-1}))]_u \\ &= [(\psi(x_0, \dots, x_{m-1}))]_{sU^n(u)} \\ &= [\psi]_{sU^n(u)} ([x_0]_{sU^n(u)}, \dots, [x_{m-1}]_{sU^n(u)}) \\ &\quad (\psi \text{ is interpreted pointwise}) \\ &= [\psi]_{sU^n(u)} ([\gamma^n x_0]_u, \dots, [\gamma^n x_{m-1}]_u) \\ &= [\psi]_u ([\gamma^n x_0]_u, \dots, [\gamma^n x_{m-1}]_u) \end{aligned}$$

because  $\psi$  is pointwise and thus, for any  $v \in \omega$ ,  $[\psi]_u = [\psi]_v$



$$= [\psi(\gamma^n(x_0), \dots, \gamma^n(x_{m-1}))]_u \quad \blacksquare$$

$$\begin{aligned} (1) \quad & [g^n \text{call}_i (\text{act}(X_0, \dots, X_{m-1}))]_u \\ &= [\text{act}(X_0, \dots, X_{m-1})]_{\text{link}(i, u, \text{stl}^n(u))} \\ &= [X_i]_{\text{dtl}(\text{link}(i, u, \text{stl}^n(u)))} \\ &= [X_i]_u \quad \blacksquare \end{aligned}$$

$$\begin{aligned} (2) \quad & [g^n \text{call}_i (\gamma X)]_u = [\gamma X]_{\text{link}(i, u, \text{stl}^n(u))} \\ &= [X]_{\text{stl}(\text{link}(i, u, \text{stl}^n(u)))} \\ &= [X]_{\text{stl}^n(u)} \\ &= [\gamma^n X]_u \quad \blacksquare \end{aligned}$$

**Theorem:** This is a generalization of (Rule 2) above.

Let  $A$  be an extensional algebra, and  $\varepsilon$  be an expression in

$\text{Florid}(A)$ . Then, for any  $i, n, m \in \omega$  such that  $n > 0$  and  $i, m \geq 0$ ,

any occurrence of  $g^m \text{call}_i \gamma^n \varepsilon$  in any  $\text{Florid}(A)$  expression can be rewritten as  $\gamma^{m+n+1} \varepsilon$

**Proof:** We use the same approach described in justifying the previous rules.

Let  $A$  be an extensional algebra.

Let  $U$  be the universe of  $\text{Flo}(A)$  (i.e.  $U = \text{bl}(\omega)$ ).

Let  $\varepsilon$  be a  $\text{Flo}(A)$ -expression,

let  $i, n, m \in \omega$  such that  $n > 0$  and  $i, m \geq 0$

then for any  $u \in U$

$$\begin{aligned} [g^m \text{call}_i \gamma^n \varepsilon]_u &= [\gamma^n \varepsilon]_{\text{link}(i, u, \text{stl}^m(u))} \\ &= [\varepsilon]_{\text{stl}^n(\text{link}(i, u, \text{stl}^m(u)))} \\ &= [\varepsilon]_{\text{stl}^{n-1}(\text{stl}(\text{link}(i, u, \text{stl}^m(u))))} \\ &= [\varepsilon]_{\text{stl}^{n-1}(\text{stl}^m(u))} \end{aligned}$$

$$= [\varepsilon]_{st|^{n-1+m}(u)}$$

$$= [\gamma^{m+n-1}\varepsilon]_u$$

■

### 5.2.2- Example

Formally speaking, a reduction sequence for a Florid-program is a sequence of Florid-expressions  $e_0, e_1, \dots, e_{n-1}$ ,

where  $e_0$  is the expression **result**; and, for every  $i$ ,  $e_i$  is a consequence of *rewriting*  $e_{i-1}$  using one of the following:

either - a rewrite rule of **Flo**,

- a rewrite rule for **A**,

or - a definition in the main Florid(A)-program.

Consider the following program in **Iswim(Q)**, where  $Q$  is the algebra of the rationals represented by their usual mathematical symbols:

**G(Y) where**

**Y = 3;**

**G(A) = F(2,A) + X + H(A) where**

**H(D) = D\*D - A + F(X,D);**

**end;**

**X = 3 + F(Z,6);**

**F(B,C) = B\*B - 2\*C ;**

**Z = 5 ;**

**end**

The program is translated into the Florid( $Q$ )-program:

$\text{result} = \text{call}_0 G ;$

$Y = 3 ;$

$G = \text{gcall}_0 F + \gamma X + \text{call}_0 H ;$

$H = D * D - \gamma A + \text{ggcall}_1 F ;$

$X = 3 + \text{call}_2 F ;$

$F = B * B - 2 * C ;$

$Z = 5 ;$

$A = \text{act}(Y) ;$

$B = \text{act}(2, X, Z) ;$

$C = \text{act}(A, D, 6) ;$

$D = \text{act}(A) ;$

Using the rewrite rules for the algebra  $\text{Flo}(Q)$  described before, we can reduce the expression  $\text{result}$  in the above set of Florid-equations as follows. For simplicity we shall break expressions into their sub-expressions; a conventional well known technique in mathematics.

$\text{result} = \text{call}_0 G$

$= \text{call}_0 (\text{gcall}_0 F + \gamma X + \text{call}_0 H) \quad (\text{Subs'n})$

$= \text{call}_0 (\text{gcall}_0 F) + \text{call}_0 \gamma X + \text{call}_0 \text{call}_0 H \quad (\text{Rule 0})$

.....(1)

$\text{call}_0 (\text{gcall}_0 F) = \text{call}_0 \text{gcall}_0 (B * B - 2 * C) \quad \text{.....(2)}$

$= \text{call}_0 \text{gcall}_0 B * \text{call}_0 \text{gcall}_0 B -$

$\text{call}_0 \text{gcall}_0 2 * \text{call}_0 \text{gcall}_0 C \quad (\text{rule 0}) \quad \text{.....(3)}$

$\text{call}_0 \text{gcall}_0 B = \text{call}_0 \text{gcall}_0 (\text{act}(2, X, Z)) \quad (\text{Subs'n})$

$= \text{call}_0 2 \quad (\text{rule 1})$

$= 2 \quad (\text{rule 0}) \quad \text{.....(4)}$

$$\begin{aligned}
\text{call}_0 (\text{gcall}_0 C) &= \text{call}_0 \text{gcall}_0 (\text{act} (A,D,6)) && (\text{Subs'n}) \\
&= \text{call}_0 A && (\text{rule 1}) \\
&= \text{call}_0 (\text{act}(Y)) && (\text{Subs'n}) \\
&= Y && (\text{rule 1}) \\
&= 3 && \dots\dots(5)
\end{aligned}$$

Therefore, by (3), (4) and (5)

$$\text{call}_0 (\text{gcall}_0 F) = 2*2 - 2*3 = -2 \quad \dots\dots(6)$$

$$\begin{aligned}
\text{call}_0 \gamma X &= X && (\text{rule 2}) \\
&= 3 + \text{call}_2 F && (\text{Subs'n}) \\
&= 3 + \text{call}_2 (B*B - 2*C) && (\text{Subs'n}) \quad \dots\dots (7) \\
\text{call}_2 B &= \text{call}_2 (\text{act}(2,X,Z)) && (\text{Subs'n}) \\
&= Z && (\text{rule 1}) \\
&= 5 && \dots\dots(8)
\end{aligned}$$

$$\begin{aligned}
\text{call}_2 C &= \text{call}_2 (\text{act} (A,D,6)) && (\text{Subs'n}) \\
&= 6 && (\text{rule 1}) \quad \dots\dots(9)
\end{aligned}$$

Therefore, by (7), (8) & (9)

$$\begin{aligned}
\text{call}_0 \gamma X &= 3 + 5*5 - 2*6 \\
&= 16 && \dots\dots(10)
\end{aligned}$$

$$\text{call}_0 \text{call}_0 H = \text{call}_0 \text{call}_0 (D*D - \gamma A + \text{ggcall}_1 F) \quad (\text{Subs'n}) \dots\dots(11)$$

$$\begin{aligned}
\text{call}_0 \text{call}_0 D &= \text{call}_0 \text{call}_0 (\text{act}(A)) && (\text{Subs'n}) \\
&= \text{call}_0 (A) && (\text{rule 1}) \\
&= \text{call}_0 (\text{act}(Y)) && (\text{Subs'n}) \\
&= Y && (\text{rule 1}) \\
&= 3 && \dots\dots(12)
\end{aligned}$$

$$\begin{aligned}
\text{call}_0 \text{call}_0 \gamma A &= \text{call}_0 (A) && (\text{rule 2}) \\
&= 3 && (\text{like in (12) above}) \quad \dots\dots(13)
\end{aligned}$$

$$\text{call}_0 \text{ call}_0 \text{ ggcall}_1 F =$$

$$\text{call}_0 \text{ call}_0 \text{ ggcall}_1 (B*B - 2*C) \text{ (Subs'n) } \dots(14)$$

$$\text{call}_0 \text{ call}_0 \text{ ggcall}_1 B =$$

$$\text{call}_0 \text{ call}_0 \text{ ggcall}_1 (\text{act}(2,X,Z)) \text{ (Subs'n) } \dots(15)$$

$$= \text{call}_0 \text{ call}_0 X \text{ (rule 1)}$$

$$= \text{call}_0 \text{ call}_0 (3 + \text{call}_2 F) \text{ (Subs'n)}$$

$$= 3 + \text{call}_0 \text{ call}_0 \text{ call}_2 F \text{ (rule 0)}$$

$$= 3 + \text{call}_0 \text{ call}_0 \text{ call}_2 (B*B - 2*C) \text{ (Subs'n) } \dots(15.1)$$

$$\text{call}_0 \text{ call}_0 \text{ call}_2 B = \text{call}_0 \text{ call}_0 \text{ call}_2 (\text{act}(2,X,Z)) \text{ (Subs'n)}$$

$$= \text{call}_0 \text{ call}_0 Z \text{ (rule 1)}$$

$$= \text{call}_0 \text{ call}_0 5 \text{ (Subs'n)}$$

$$= 5 \dots(15.2)$$

$$\text{call}_0 \text{ call}_0 \text{ call}_2 C = \text{call}_0 \text{ call}_0 \text{ call}_2 (\text{act}(A,D,6)) \text{ (Subs'n)}$$

$$= 6 \dots(15.3)$$

Therefore, by 15, 15.1, 15.2, and 15.3

$$\text{call}_0 \text{ call}_0 \text{ ggcall}_1 B = 3 + 5*5 - 2*6$$

$$= 16 \dots(16)$$

$$\text{call}_0 \text{ call}_0 \text{ ggcall}_1 C = \text{call}_0 \text{ call}_0 \text{ ggcall}_1 (\text{act}(A,D,6))$$

$$= \text{call}_0 \text{ call}_0 D \text{ (rule 1)}$$

$$= \text{call}_0 \text{ call}_0 (\text{act}(A)) \text{ (Subs'n)}$$

$$= \text{call}_0 A \text{ (rule 1)}$$

$$= \text{call}_0 (\text{act}(Y)) \text{ (Subs'n)}$$

$$= Y \text{ (rule 1)}$$

$$= 3 \dots(17)$$

Therefore by 14, 16, and 17



$$\begin{aligned}\text{call}_0 \text{call}_0 \text{ggcall}_1 F &= \text{call}_0 \text{call}_0 \text{ggcall}_1 (B*B - 2*C) \\ &= 16*16 - 2*3 = 250 \quad \dots(18)\end{aligned}$$

By 11, 12, 13, and 18

$$\begin{aligned}\text{call}_0 \text{call}_0 H &= \text{call}_0 \text{call}_0 (D*D - \gamma A + \text{ggcall}_1 F) \text{ (Subs'n)} \\ &= 3*3 - 3 + 250 = 256 \quad \dots(19)\end{aligned}$$

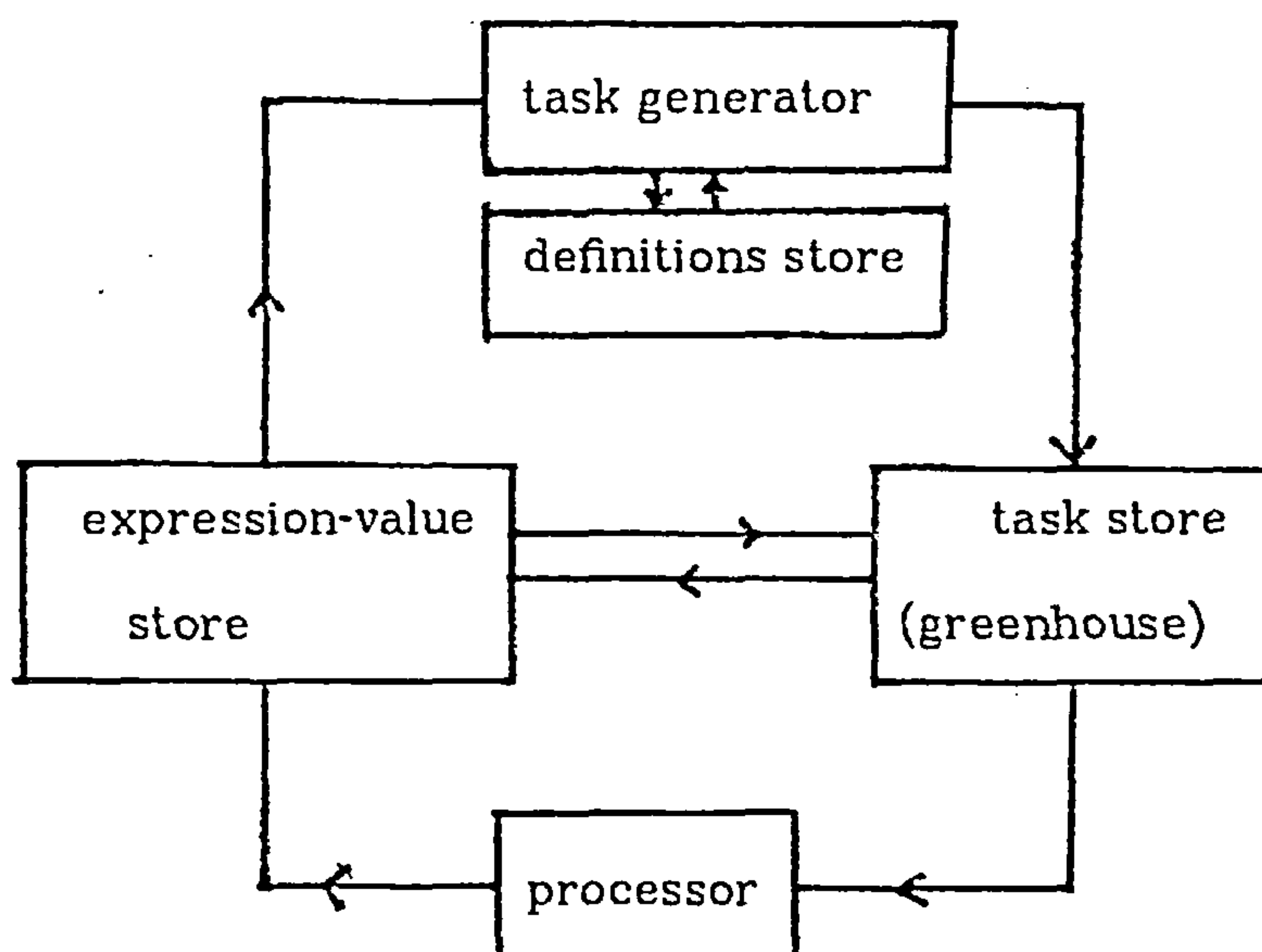
Therefore, by 1, 6, 10, and 19

$$\begin{aligned}\text{result} &= \text{call}_0 (\text{gcall}_0 F) + \text{call}_0 \gamma X + \text{call}_0 \text{call}_0 H \\ &= -2 + 16 + 256 = 270\end{aligned}$$

### 5.3- Tagged Education for Florid:

In this section, we describe the tagged education approach to evaluating the target family Florid. Our interest is mainly with the schema of evaluation rather than with the hardware of the machine. The architectural consideration of such a machine is beyond the scope of this thesis. This schema can be used, also, to implement Florid on a conventional machine, however it lends itself directly to tagged education machines such as the Tagged Education Engine [Ash84].

In Chapter 0, we have described a simple tagged education machine, and demonstrated the education process of evaluating expressions in Basic Lucid. Let us consider the same simple machine; it is diagrammed as



The value of an expression in Basic Lucid is an infinite sequence of objects, each element in the sequence is the value of the expression at a certain moment in time. Therefore, when giving a certain expression to the processor we should supply the processor with the moment in time at which we require the value of the expression.

Thus, a data token (or a daton) in the education of Basic Lucid should consist of two cells; the expression cell which contains the expression to be evaluated, and the tag cell which contains the moment in time at which the value of the

expression is required.

For example, if we want the fourth element in the sequence value of the expression  $\text{first}(X+Y)$  then the tagged daton will look like <sup>†</sup>

expression	tag
$\text{first}(X+Y)$	3

When this daton is received by the processor, the latter demands the value of the expression  $X+Y$  at time 0. We can think, at this moment, that the original task of evaluating the value of  $\text{first}(X+Y)$  at time 3 is sent back to the greenhouse to wait for its argument to be computed (we shall say that a task is ripen when all its arguments are computed). The demand for  $X+Y$  at time 0 is received by the expression-value store which checks whether the required value is stored there. If the value is there, then it is sent to the greenhouse to ripen the original task which at this moment becomes ready to be passed to the processor. Otherwise, the demand for such a value is sent to the task generator which generates a new task to evaluate such a value; i.e. it generates a task for evaluating the tagged daton

$X+Y$	0
-------	---

When such a task is received by the processor, it is returned to the greenhouse and the processor demands the values of the datons

$X$	0	and	$Y$	0
-----	---	-----	-----	---

Assuming that such values do exist in the expression-value store, they will be sent to the greenhouse. The task for evaluating the initial value of  $X+Y$  is now

<sup>†</sup> The reader is reminded that the  $n$ th value of an expression is its value at time  $n-1$  because its first value is that at time 0.

ripen (what the processor has demanded for its evaluation are now available) and will be passed to the processor. The latter evaluates

X+Y	0
-----	---

and returns the value to the expression-value store. The original task of evaluating the value of  $\text{first}(X+Y)$  at time 3 is now ripen and sent to the processor which will return the initial value of  $X+Y$ .

Let us denote the result of demanding the value of the Basic Lucid expression  $X$  at time  $n$  as  $\text{Dem}(X, n)$ . Then the eduction process described above can be denoted by the following equations

$$\text{Dem}(\text{first}(X+Y), 3) = \text{Dem}(X+Y, 0) = \text{Dem}(X, 0) \div \text{Dem}(Y, 0)$$

The language  $\text{Florid}(A)$ , for an extensional algebra  $A$ , is  $DE(\text{Flo}(A))$ . The eduction process for evaluating  $\text{Florid}(A)$  expressions is very similar to that described for Basic Lucid. The main difference is in the structure of the tagged datons. Since the universe of the algebra  $Lu$  is the set  $\omega$  the tag cell of a daton in the eduction of Basic Lucid is a non-negative integer (representing a moment in time). In the eductive evaluation of  $\text{Florid}$  the expressions should be tagged with b-lists because the universe of  $\text{Flo}(A)$  is the set  $\text{bl}(\omega)$  or the set of lists with back pointers over the natural numbers.

Thus, a daton in the evaluation of  $\text{Florid}$  consists of two cells; the expression cell which contains the  $\text{Florid}$ -expression to be evaluated and the tag cell which contains a b-list representing a point in the tree of function calls.

expression	tag: b-list
------------	-------------

a daton (tagged expression) in the eduction of  $\text{Florid}$



Suppose we want to evaluate the Florid(N)-expression  $\text{act}(z,3,x,y)$  at the b-list  $\langle 3,\alpha,\beta \rangle$ , where  $\langle 3,\alpha,\beta \rangle$  is the b-list whose head is 3, dynamic tail is  $\alpha$  and static tail is  $\beta$ . The evaluation process can be represented by the following scenario between the processor and the other modules of the eduction machine:

The processor is given the task of evaluating the daton

$\text{act}(z,3,x,y)$	$\langle 3,\alpha,\beta \rangle$
-----------------------	----------------------------------

The processor demands the value of the daton 

$y$	$\alpha$
-----	----------

Given such a value, the processor returns it as the result of the original task.

Using the meta-function Dem, the above scenario can be denoted by

$$\text{Dem}(\text{act}(z,3,x,y), \langle 3,\alpha,\beta \rangle) = \text{Dem}(y, \alpha)$$

Basically, there are two classes of basic operations which have to be performed by the processor. The first corresponds to the pointwise operators in Florid(A); that is, the pointwise extension of the operators of the algebra A. For example, in Florid(N) where N is the algebra over the natural numbers

$$\text{Dem}(X+Y, \alpha) = \text{Dem}(X, \alpha) + \text{Dem}(Y, \alpha)$$

$$\text{Dem}(X-Y, \alpha) = \text{Dem}(X, \alpha) - \text{Dem}(Y, \alpha)$$

$$\text{Dem}(X*Y, \alpha) = \text{Dem}(X, \alpha) * \text{Dem}(Y, \alpha)$$

and so on.

Notice that, if  $x$  is a constant, then for any  $\alpha \in \text{bl}(\omega)$

$$\text{Dem}(x, \alpha) = x$$

The second class of operators are the intensional operators defined by the algebra *Flo*. From the definition of *Flo*, we can specify the behaviour of the eduction processor in the following equations:

$$\text{Dem}(\gamma X, \alpha) = \text{Dem}(X, \text{stl}(\alpha))$$



$$\text{Dem}(\text{act}(x_0, \dots, x_{n-1}), \alpha) = \text{Dem}(X(\text{hd}(\alpha)), \text{dtl}(\alpha))$$

$$\text{Dem}(\text{call}_i X, \alpha) = \text{Dem}(X, \beta)$$

$$\text{where } \beta = \langle i, \alpha, \alpha \rangle (= \text{link}(i, \alpha, \alpha))$$

$$\text{Dem}(\text{gcall}_i X, \alpha) = \text{Dem}(X, \beta)$$

$$\text{where } \beta = \langle i, \alpha, \text{stl}(\alpha) \rangle (= \text{link}(i, \alpha, \text{stl}(\alpha)))$$

and generally

$$\text{Dem}(\mathbf{g}^n \text{call}_i, \alpha) = \text{Dem}(X, \beta)$$

$$\text{where } \beta = \langle i, \alpha, \text{stl}^n(\alpha) \rangle$$

#### 5.4- Example:

We consider here the same program discussed in section 5.2.2. The  $\text{Iswim}(Q)$  program

$G(Y)$  where

$Y = 3 ;$

$G(A) = F(2,A) + X + H(A)$  where

$H(D) = D * D - A + F(X,D) ;$

end;

$X = 3 + F(Z,6);$

$F(B,C) = B * B - 2 * C ;$

$Z = 5 ;$

end

is translated into the  $\text{Florid}(Q)$  program

$\text{result} = \text{call}_0 G ;$

$Y = 3$

$G = g\text{call}_0 F + \gamma X + \text{call}_0 H ;$

$H = D * D - \gamma A + g\text{gcall}_1 F$

$X = 3 + \text{call}_2 F ;$

$F = B * B - 2 * C ;$

$Z = 5 ;$

$A = \text{act}(Y) ;$

$B = \text{act}(2, X, Z) ;$

$C = \text{act}(A, D, 6) ;$

$D = \text{act}(A)$

We use here the meta-function  $\text{Dem}$  to describe the eductive evaluation of the program.

From the semantics of  $\text{Florid}$ , the value of the program is the value of  $\text{result}$

at the empty b-list  $\Lambda$ . Therefore, we start with the value  $\text{Dem}(\text{result}, \Lambda)$

$$\begin{aligned}
 \text{Dem}(\text{result}, \Lambda) &= \text{Dem}(\text{call}_0 G, \Lambda) \\
 &= \text{Dem}(G, \langle 0, \Lambda, \Lambda \rangle) \\
 &= \text{Dem}(\text{gcall}_0 F + \gamma X + \text{call}_0 H, \langle 0, \Lambda, \Lambda \rangle) \\
 &= \text{Dem}(\text{gcall}_0 F, \langle 0, \Lambda, \Lambda \rangle) + \text{Dem}(\gamma X, \langle 0, \Lambda, \Lambda \rangle) + \\
 &\quad \text{Dem}(\text{call}_0 H, \langle 0, \Lambda, \Lambda \rangle) \quad (1)
 \end{aligned}$$

$$\begin{aligned}
 \text{Dem}(\text{gcall}_0 F, \langle 0, \Lambda, \Lambda \rangle) &= \text{Dem}(F, \langle 0, \beta, \Lambda \rangle) \quad \text{where } \beta = \langle 0, \Lambda, \Lambda \rangle \\
 &= \text{Dem}(B*B - 2*C, \langle 0, \beta, \Lambda \rangle) \\
 &= \text{Dem}(B, \langle 0, \beta, \Lambda \rangle) * \text{Dem}(B, \langle 0, \beta, \Lambda \rangle) - \text{Dem}(2, \langle 0, \beta, \Lambda \rangle) * \\
 &\quad \text{Dem}(C, \langle 0, \beta, \Lambda \rangle) \quad (2)
 \end{aligned}$$

$$\begin{aligned}
 \text{Dem}(B, \langle 0, \beta, \Lambda \rangle) &= \text{Dem}(\text{act}(2, X, Z), \langle 0, \beta, \Lambda \rangle) \\
 &= \text{Dem}(2, \beta) = 2 \quad (3)
 \end{aligned}$$

$$\begin{aligned}
 \text{Dem}(C, \langle 0, \beta, \Lambda \rangle) &= \text{Dem}(\text{act}(A, D, 6), \langle 0, \beta, \Lambda \rangle) \\
 &= \text{Dem}(A, \beta) \\
 &= \text{Dem}(\text{act}(3), \langle 0, \Lambda, \Lambda \rangle) \\
 &= \text{Dem}(3, \Lambda) = 3 \quad (4)
 \end{aligned}$$

$$\begin{aligned}
 \text{Hence } \text{Dem}(\text{gcall}_0 F, \langle 0, \Lambda, \Lambda \rangle) &= 2*2 - 2*3 \\
 &= -2 \quad (\text{by } 2, 3, 4) \quad (5)
 \end{aligned}$$

$$\begin{aligned}
 \text{Dem}(\gamma X, \langle 0, \Lambda, \Lambda \rangle) &= \text{Dem}(X, \Lambda) \\
 &= \text{Dem}(3 + \text{call}_2 F, \Lambda) \\
 &= \text{Dem}(3, \Lambda) + \text{Dem}(\text{call}_2 F, \Lambda) \quad (6)
 \end{aligned}$$

$$\begin{aligned}
 \text{Dem}(\text{call}_2 F, \Lambda) &= \text{Dem}(F, \langle 1, \Lambda, \Lambda \rangle) \\
 &= \text{Dem}(B*B - 2*C, \langle 1, \Lambda, \Lambda \rangle) \quad (7)
 \end{aligned}$$

$$\begin{aligned} \text{Dem}(B, \langle 2, \Lambda, \Lambda \rangle) &= \text{Dem}(\text{act}(2, X, Z), \langle 2, \Lambda, \Lambda \rangle) \\ &= \text{Dem}(Z, \Lambda) = 5 \end{aligned} \quad (8)$$

$$\begin{aligned} \text{Dem}(C, \langle 2, \Lambda, \Lambda \rangle) &= \text{Dem}(\text{act}(A, D, 6), \langle 2, \Lambda, \Lambda \rangle) \\ &= \text{Dem}(6, \Lambda) = 6 \end{aligned} \quad (9)$$

$$\begin{aligned} \text{Hence } \text{Dem}(\gamma X, \langle 0, \Lambda, \Lambda \rangle) &= 3 + 5 * 5 - 2 * 6 \\ &= 16 \quad (\text{by } 6, 7, 8, 9) \end{aligned} \quad (10)$$

$$\begin{aligned} \text{Dem}(\text{call}_0 H, \langle 0, \Lambda, \Lambda \rangle) &= \text{Dem}(H, \langle 0, \vartheta, \vartheta \rangle) \\ &\quad \text{where } \vartheta = \langle 0, \Lambda, \Lambda \rangle \\ &= \text{Dem}(D * D - \gamma A + \text{ggcall}_1 F, \langle 0, \vartheta, \vartheta \rangle) \\ &= \text{Dem}(D, \langle 0, \vartheta, \vartheta \rangle) * \text{Dem}(D, \langle 0, \vartheta, \vartheta \rangle) \\ &\quad - \text{Dem}(\gamma A, \langle 0, \vartheta, \vartheta \rangle) + \text{Dem}(\text{ggcall}_1 F, \langle 0, \vartheta, \vartheta \rangle) \end{aligned} \quad (11)$$

$$\begin{aligned} \text{Dem}(D, \langle 0, \vartheta, \vartheta \rangle) &= \text{Dem}(\text{act}(A), \langle 0, \vartheta, \vartheta \rangle) \\ &= \text{Dem}(A, \vartheta) \\ &= \text{Dem}(\text{act}(Y), \langle 0, \Lambda, \Lambda \rangle) \\ &= \text{Dem}(Y, \Lambda) \\ &= \text{Dem}(3, \Lambda) = 3 \end{aligned} \quad (12)$$

$$\begin{aligned} \text{Dem}(\gamma A, \langle 0, \vartheta, \vartheta \rangle) &= \text{Dem}(\Lambda, \vartheta) \\ &= \text{Dem}(\text{act}(Y), \langle 0, \Lambda, \Lambda \rangle) \\ &= 3 \quad (\text{as in } 12) \end{aligned} \quad (13)$$

$$\begin{aligned} \text{Dem}(\text{ggcall}_1 F, \langle 0, \vartheta, \vartheta \rangle) &= \text{Dem}(F, \langle 1, \langle 0, \vartheta, \vartheta \rangle, \Lambda \rangle) \\ &= \text{Dem}(B, \langle 1, \langle 0, \vartheta, \vartheta \rangle, \Lambda \rangle) * \text{Dem}(B, \langle 1, \langle 0, \vartheta, \vartheta \rangle, \Lambda \rangle) \\ &\quad * \text{Dem}(2, \langle 1, \langle 0, \vartheta, \vartheta \rangle, \Lambda \rangle) * \text{Dem}(C, \langle 1, \langle 0, \vartheta, \vartheta \rangle, \Lambda \rangle) \end{aligned} \quad (14)$$

$$\begin{aligned} \text{Dem}(B, \langle 1, \langle 0, \vartheta, \vartheta \rangle, \Lambda \rangle) &= \text{Dem}(\text{act}(2, X, Z), \langle 1, \langle 0, \vartheta, \vartheta \rangle, \Lambda \rangle) \\ &= \text{Dem}(X, \langle 0, \vartheta, \vartheta \rangle) \end{aligned}$$

$$\begin{aligned}
&= \text{Dem}(3 + \text{call}_2 F, \langle 0, v, v \rangle) \\
&= 3 + \text{Dem}(\text{call}_2 F, \langle 0, v, v \rangle) \\
&= 3 + \text{Dem}(F, \langle 2, \langle 0, v, v \rangle, \langle 0, v, v \rangle \rangle) \\
&= 3 + \text{Dem}(B*B - 2*C, \langle 2, \langle 0, v, v \rangle, \langle 0, v, v \rangle \rangle) \quad (15)
\end{aligned}$$

$$\begin{aligned}
&\text{Dem}(B, \langle 2, \langle 0, v, v \rangle, \langle 0, v, v \rangle \rangle) \\
&= \text{Dem}(\text{act}(2, X, Z), \langle 2, \langle 0, v, v \rangle, \langle 0, v, v \rangle \rangle) \\
&= \text{Dem}(Z, \langle 0, v, v \rangle) \\
&= \text{Dem}(5, \langle 0, v, v \rangle) = 5 \quad (16)
\end{aligned}$$

$$\begin{aligned}
&\text{Dem}(C, \langle 2, \langle 0, v, v \rangle, \langle 0, v, v \rangle \rangle) \\
&= \text{Dem}(\text{act}(A, D, 6), \langle 2, \langle 0, v, v \rangle, \langle 0, v, v \rangle \rangle) \\
&= \text{Dem}(6, \langle 0, v, v \rangle) = 6 \quad (17)
\end{aligned}$$

$$\begin{aligned}
\text{Therefore } \text{Dem}(B, \langle 1, \langle 0, v, v \rangle, \Lambda \rangle) &= 3 + 5*5 - 2*6 \\
&= 16 \quad (\text{by } 15, 16, 17) \quad (18)
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(C, \langle 1, \langle 0, v, v \rangle, \Lambda \rangle) &= \text{Dem}(\text{act}(A, D, 6), \langle 1, \langle 0, v, v \rangle, \Lambda \rangle) \\
&= \text{Dem}(D, \langle 0, v, v \rangle) \\
&= \text{Dem}(\text{act}(\Lambda), \langle 0, v, v \rangle) \\
&= \text{Dem}(A, v) \\
&= \text{Dem}(\text{act}(Y), \langle 0, \Lambda, \Lambda \rangle) \\
&= \text{Dem}(Y, \Lambda) \\
&= \text{Dem}(3, \Lambda) = 3 \quad (19)
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\text{ggcall}_1 F, \langle 0, v, v \rangle) &= 16*16 - 2*3 \\
&= 250 \quad (\text{by } 14, 18, 19) \quad (20)
\end{aligned}$$

$$\begin{aligned}
\text{Dem}(\text{call}_0 H, \langle 0, \Lambda, \Lambda \rangle) &= 3*3 - 3 + 250 \\
&= 256 \quad (\text{by } 11, 12, 13, 20) \quad (21)
\end{aligned}$$

$$\begin{aligned}
\text{Thus, } \text{Dem}(\text{result}, \Lambda) &= -2 + 16 + 256 = 270 \\
&(\text{by } 1, 5, 10, 21)
\end{aligned}$$



## Chapter 6

### The Compilation of Luswim

#### 6.0- Introduction:

Thus far, we have described the compilation of Lambda calculus-based first order functional languages or, in particular, languages of the ISWIM family in which all functions are of first order. Apart from their importance as programming languages on their own right, our interest in these languages is due to the fact that Lucid is based on this family. Lucid is the hybrid of this family with iteration [WaAs83, AsWa80]. In this chapter, we describe both the compilation process and the evaluation techniques for the family Luswim.

#### 6.1- *Flu* : The Algebra of Place and Time

Since the language lswim has been compiled into Florid which is the family  $DE(Fl_0)$ ; we expect Luswim, which is lswim( $L_u$ ), to be compiled into  $DE(Fl_0 \circ L_u)$ . If we think of the construction of the intensional algebra  $Fl_0 \circ L_u$  as applying the algebra  $Fl_0$  on  $L_u$ , then we end up applying  $Fl_0$  on intensional algebras which is against the definition of  $Fl_0$ . Moreover, it tends to be cumbersome as we have to think about infinite families of infinite families of data objects. Notice that, in such a construction method, the intensional domain for  $Fl_0(L_u(A))$ , for an extensional data algebra  $A$ , is  $[U \rightarrow {}^\omega D]$  where  $D$  is the domain of  $A$ .

Instead, we join the families  $Fl_0$  and  $L_u$  in the family of intensional algebras  $Fl_u$ . The universe of  $Fl_u$  is the cartesian product of the universes of  $Fl_0$  and  $L_u$ . Moreover,  $Fl_u$  preserves the meaning of both  $Fl_0$  and  $L_u$ ; that is, it extends the meaning of each algebra pointwise over the cartesian product

of their universes.

It is not difficult to see, from our definition of  $Flu$  that it is equivalent (up to isomorphism) to  $Flo \circ Lu$ , but simpler and clearer.

On one hand, the universe of the algebra  $Flo$  was defined to be the set  $bl(\omega)$ ; i.e. the set of lists with back pointers over the natural numbers. Each of these lists represent a **place** in the tree of function calls. On the other hand, the universe of  $Lu$  is the set of natural numbers  $\omega$  representing instances in time. Thus, the universe of  $Flu$  is the set  $bl(\omega) \times \omega$ . When one thinks of a world as a pair representing a place (or a function call) together with an instance in time, the reason for the above terminology, i.e. the algebra of time and place, becomes clear.

**Definition (The algebra  $Flu$ ):**

Let  $A (= \langle D, F \rangle)$  be an extensional  $\Sigma$ -algebra.

Then,  $Flu(A)$  is the intensional  $\Sigma$ -algebra whose universe is  $bl(\omega) \times \omega$ , and such that

a:  $Flu(A)$  extends the operations of  $A$  pointwise over the universe

$bl(\omega) \times \omega$ . That is,

for every  $n$ -ary constant symbol  $\psi$  in  $\Sigma$ ,

for every  $n$   $Flu(A)$ -terms  $x_0, \dots, x_{n-1}$ ,

and  $\forall \langle u, i \rangle \in bl(\omega) \times \omega$

$$(Flu(\psi)(x_0, \dots, x_{n-1}))_{\langle u, i \rangle} = A(\psi)((Flu(x_0))_{\langle u, i \rangle}, \dots, (Flu(x_{n-1}))_{\langle u, i \rangle})$$

b:  $Flu(A)$  preserves the meaning of  $Lu(A)$ . That is,

for every  $n$ -ary Lucid operator symbol  $\vartheta$

for the  $Flu(A)$ -terms  $x_0, \dots, x_{n-1}$ , and  $\forall \langle u, i \rangle \in bl(\omega) \times \omega$

$$(Flu(\vartheta)(x_0, \dots, x_{n-1}))_{\langle u, i \rangle} = Lu(\vartheta)_i((Flu(x_0))_{\langle u, i \rangle}, \dots, (Flu(x_{n-1}))_{\langle u, i \rangle})$$

Therefore

$$(Flu(\text{first}(X)))_{\langle u,i \rangle} = (Flu(X))_{\langle u,0 \rangle}$$

$$(Flu(\text{next}(X)))_{\langle u,i \rangle} = (Flu(X))_{\langle u,i+1 \rangle}$$

$$(Flu(\text{fby}(X,Y)))_{\langle u,i \rangle} = \text{if } i \text{ eq } 0 \text{ then } (Flu(X))_{\langle u,i \rangle} \\ \text{else } (Flu(Y))_{\langle u,i-1 \rangle}$$

$$(Flu(\text{whenever}(X,Y)))_{\langle u,i \rangle} = \text{if } (Flu(Y))_{\langle u,i \rangle} \text{ then } (Flu(X))_{\langle u,i \rangle} \\ \text{else } \perp$$

$$(Flu(\text{asa}(X,Y)))_{\langle u,i \rangle} = \text{if } \exists j \leq i \text{ such that}$$

$$(Flu(Y))_{\langle u,j \rangle} \text{ and } (\forall k \in \omega \ k < j \rightarrow \text{not } (Flu(Y))_{\langle u,k \rangle})$$

$$\text{then } (Flu(X))_{\langle u,j \rangle} \text{ else } \perp$$

$$(Flu(\text{upon}(X,Y)))_{\langle u,i \rangle} = (Flu(X))_{\langle u,n \rangle}$$

$$\text{where } n = \text{Card}\{j \leq i : (Flu(Y))_{\langle u,j \rangle}\}$$

and Card is the cardinality function on sets

$$(Flu(\text{attime}(X,Y)))_{\langle u,i \rangle} = (Flu(X))_{\langle u,n \rangle}$$

$$\text{where } n = (Flu(Y))_{\langle u,i \rangle}$$

c:  $Flu(A)$  preserves the meaning of  $Flo(A)$ . That is,

for every  $n$ -ary operator symbol  $\psi$  in the signature of  $Flo$ ,

for every  $Flu(A)$ -terms  $x_0, \dots, x_{n-1}$ , and  $\forall \langle u,i \rangle \in \text{bl}(\omega) \times \omega$

$$(Flu(\psi)(x_0, \dots, x_{n-1}))_{\langle u,i \rangle} = Flo(\psi)_u((Flu(x_0))_{\langle u,i \rangle}, \dots, (Flu(x_{n-1}))_{\langle u,i \rangle})$$

Therefore

$$(Flu(\text{act}(x_0, \dots, x_{j-1})))_{\langle u,i \rangle} = (Flu(x_{hd(u)}))_{\langle dtl(u),i \rangle}$$

$$(Flu(\gamma X))_{\langle u,i \rangle} = (Flu(X))_{\langle stl(u),i \rangle}$$

$$(Flu(\gamma\gamma X))_{\langle u,i \rangle} = (Flu(X))_{\langle stl(stl(u)),i \rangle}$$

and generally, for any  $n \in \omega$ ,

$$(Flu(\gamma^n X))_{\langle u,i \rangle} = (Flu(X))_{\langle stl^n(u),i \rangle}$$

For any  $j \in \omega$

$$(Flu(\text{call}_j X))_{\langle u,i \rangle} = (Flu(X))_{\langle v,i \rangle} \quad \text{where } v = \langle j, u, u \rangle (= \text{link}(j, u, u))$$



$$(Flu(gcall_j X))_{\langle u,i \rangle} = (Flu(X))_{\langle v,i \rangle} \quad \text{where } v = \langle j, u, stl(u) \rangle$$

and generally, for any  $n \in \omega$ ,

$$(Flu(g^n call_j X))_{\langle u,i \rangle} = (Flu(X))_{\langle v,i \rangle} \quad \text{where } v = \langle j, u, stl^n(u) \rangle$$

Given an extensional  $\Sigma$ -algebra  $A$ , the intersection of the signature of  $Flo(A)$  and that of  $Lu(A)$  is only  $\Sigma$ . Therefore, the algebra  $Flu$  given above is well defined (i.e. a function).

Moreover, it is not difficult to see from the definition of  $Flu(A)$  that it is isomorphic to  $Flo(Lu(A))$ . In fact it is just a curried version; instead of interpreting an  $n$ -ary operator symbol as a function in

$[(U \rightarrow {}^\omega D))^n \rightarrow [U \rightarrow {}^\omega D]]$  we consider it as a function in

$$[(U \times {}^\omega D))^n \rightarrow [U \times {}^\omega D]]$$

## 6.2- The Target Language Fluid for Luswim:

The target language Florid( $A$ ) was defined to be  $DE(Flo(A))$ ; it is an equational language over the intensional terms of  $Flo(A)$ . That is, a program is a set of equations defining nullary variable symbols and the expressions (the terms) are those constructed from the operator symbols of the algebra  $Flo(A)$ .

In this section, we define the family of languages **Fluid** which will be the target for the family Luswim. Fluid is the family

$$\{DE(Flu(A)) \mid A \text{ is an extensional algebra}\}.$$

That is, given an extensional algebra  $A$ , the member Luswim( $A$ ) is translated into Fluid( $A$ ); i.e. into  $DE(Flu(A))$ .

**Fluid**, like **Florid**, is an equational language over intensional terms. However, the intensional terms are richer as they are defined by the algebra  $Flu$  and hence includes terms generated by the constant symbols of both  $Flo$  and  $Lu$ .

For an extensional algebra  $A$ , we want to translate the language  $Luswim(A)$  which is  $Iswim(Lu(A))$  into  $Fluid(A)$  which is  $DE(Flu(A))$ . It is important to remember here that the constant symbols of  $Lu$  are also in the signature of  $Flu$ ; and thus the terms generated by the constant symbols of the algebra  $Lu$  are also terms in the target algebra  $Flu$ . Therefore, the translation should preserve those terms; that is, the translation algorithm should pass them as they are.

The translation algorithm from  $Luswim$  into  $Fluid$ , then, will be the very same algorithm for compiling  $Iswim$  into  $Fluid$  described in 4.8.1.

The following example demonstrates the translation from  $Luswim$  into  $Fluid$ .

**Example:**

The following is a program in  $Plucid$ , which is a member of the  $Lucid$  family developed at Warwick [FMY 83] [WaAs84]. The data types of  $Plucid$  are those of  $POP2$  together with the objects `error` and `eod` (for end of data). We would like to mention here that `iscod` is an operator in  $Plucid$  whose value at any point  $i \in \omega$  is true when the corresponding value of its argument is the `eod` object. The only user defined functions in the program are `qsort` and `insert`.

The program takes a sequence of input values terminated by the object `eod`, and returns the sequence sorted in an ascending order. The text, in the program preceded by `//` are comments.



qsort(input) where

```
qsort(x) = if iscod(nextelement) then x
          else insert(v,z) fi
```

where

```
v = qsort(low);
```

```
z = first x fby qsort(hi) ;
```

```
nextelement = first (next(x));
```

```
low = next x wvr next x < first x;
```

```
// all the elements below x
```

```
hi = next x wvr first x <= next x;
```

```
// all the elements above x
```

```
insert(a,b) = if not iscod (a) then a
```

```
            else b upon iscod (a) fi;
```

```
// insert x between low and the hi
```

```
end; end
```

The Fluid program is:

```
result = call0 qsort
```

```
qsort = if iscod(nextelement) then x else call0 insert fi
```

```
v = call1 qsort
```

```
z = first x fby call2 qsort
```

```
nextelement = first next x
```

```
low = next x wvr next x < first x
```

```
hi = next x wvr first x <= next x
```

```
insert = if not iscod(a) then a else b upon iscod(a) fi
```

```
x = act (input,low,hi)
```

```
a = act (v)
```

```
b = act (z)
```

### 6.3 Rewrite Rules for *Flu*

As *Flu* is the join of *Lu* and *Flo*, then the rewrite rules of *Flu* consist of those of *Lu* together with those of *Flo*. The only new rule, besides those rules, is the **distribution Rule** which states that the operations of *Flo* are distributive over those of *Lu*. In this section, we state these rules and give a justification only for the distribution rule. Justifying the other rules is straight forward and analogous to the proofs of section 5.2.1 and those given in this section. We leave them as an exercise to the interested reader.

#### The Rules of *Flo*

##### (Rule *Flo* 0)

Let  $A$  be an extensional  $\Sigma$ -algebra, and let  $\psi$  be an  $n$ -ary symbol in  $\Sigma$ . Then for the *Flu*( $A$ )-expressions

$x_0, \dots, x_{n-1}$ , and for every operator symbol  $\vartheta$  in the signature of *Flo*

$$\vartheta(\psi(x_0, \dots, x_{n-1})) = \psi(\vartheta(x_0), \dots, \vartheta(x_{n-1}))$$

That is, the operators of *Flo* are distributive over those of the extensional algebra  $A$ .

If  $X, X_0, \dots, X_{m-1}$  are *Flo*( $A$ )-terms, then

(Rule *Flo* 1) For any  $i, n, m \in \omega$   $g^n \text{call}_i(\text{act}(X_0, \dots, X_{m-1})) = X_i$

(Rule *Flo* 2) For any  $i, n \in \omega$   $g^n \text{call}_i(\gamma X) = \gamma^n X$

#### The rules of *Lu*

Let  $A$  be an extensional  $\Sigma$ -algebra. Then

(Rule *Lu* 0) **first** and **next** are distributive over the operations

of  $A$ . That is, for every  $n$ -ary symbol  $\psi$  in  $\Sigma$ , and the

expressions  $x_0, \dots, x_{n-1}$  in  $\mathcal{Flu}(A)$ ,

$$\text{first}(\psi(x_0, \dots, x_{n-1})) = \psi(\text{first}(x_0), \dots, \text{first}(x_{n-1}))$$

$$\text{next}(\psi(x_0, \dots, x_{n-1})) = \psi(\text{next}(x_0), \dots, \text{next}(x_{n-1}))$$

(Rule  $\mathcal{Lu}1$ ) For every expressions  $X, Y$  in  $\mathcal{Flu}(A)$

$$\text{first}(X \text{ fby } Y) = \text{first}(X)$$

$$\text{next}(X \text{ fby } Y) = Y$$

The Distribution Rule (DR):

If  $\psi$  is an  $m$ -ary operator symbol in the signature of  $\mathcal{Lu}$ ,

and  $x_0, \dots, x_{m-1}$  are  $m$  expressions in  $\mathcal{Flu}(A)$ .

Then  $\forall i, n \in \omega$

$$g^n \text{call}_i(\psi(x_0, \dots, x_{m-1})) = \psi(g^n \text{call}_i x_0, \dots, g^n \text{call}_i x_{m-1})$$

$$\gamma^n(\psi(x_0, \dots, x_{m-1})) = \psi(\gamma^n x_0, \dots, \gamma^n x_{m-1})$$

That is,  $\forall i, n \in \omega$

$$(DR1) \quad g^n \text{call}_i(\text{first}(X)) = \text{first}(g^n \text{call}_i(X))$$

$$\gamma^n(\text{first}(X)) = \text{first}(\gamma^n(X))$$

$$(DR2) \quad g^n \text{call}_i(\text{next}(X)) = \text{next}(g^n \text{call}_i(X))$$

$$\gamma^n(\text{next}(X)) = \text{next}(\gamma^n(X))$$

$$(DR3) \quad g^n \text{call}_i(X \text{ fby } Y) = (g^n \text{call}_i X) \text{ fby } (g^n \text{call}_i Y)$$

$$\gamma^n(X \text{ fby } Y) = (\gamma^n X) \text{ fby } (\gamma^n Y)$$

$$(DR4) \quad g^n \text{call}_i(X \text{ wvr } Y) = (g^n \text{call}_i X) \text{ wvr } (g^n \text{call}_i Y)$$

$$\gamma^n(X \text{ wvr } Y) = (\gamma^n X) \text{ wvr } (\gamma^n Y)$$

and the same for  $\text{asa}$ ,  $\text{upon}$ , and all other Lucid operators.

**Proof:** To justify a rewrite rule, we have to show that the rule (the equality) holds in any world in the universe. For example, if  $A = B$  is a rule, then we have

to show that the value of  $A$  in any world is equivalent to the value of  $B$  in that same world. For simplicity, we denote the value of an expression  $X$  in a world  $v$  by  $[X]_v$ .

Therefore, if  $A = B$  is a rewrite rule for  $Flu$ , we have to show that for any world  $\langle v, i \rangle$  in the universe of  $Flu$ , which is  $bl(\omega) \times \omega$

$$[A]_{\langle v, i \rangle} = [B]_{\langle v, i \rangle}$$

We prove here three cases of the distribution rule. All other cases are almost the same

Let  $\langle u, i \rangle \in bl(\omega) \times \omega$ , and let  $n \in \omega$

(DR1) prove that  $g^n call_i (first(X)) = first(g^n call_i (X))$

$$\begin{aligned} [g^n call_i (first(X))]_{\langle u, i \rangle} &= [first(X)]_{\langle link(i, u, stl^n(u)), i \rangle} \\ &= [X]_{\langle link(i, u, stl^n(u)), 0 \rangle} \\ &= [g^n call_i X]_{\langle u, 0 \rangle} \\ &= [first(g^n call_i X)]_{\langle u, i \rangle} \end{aligned}$$

■

(DR3) Prove that  $g^n call_i (X fby Y) = (g^n call_i X) fby (g^n call_i Y)$

$$\begin{aligned} [g^n call_i (X fby Y)]_{\langle u, i \rangle} &= [(X fby Y)]_{\langle link(i, u, stl^n(u)), i \rangle} \\ &= \lambda i \in \omega . \text{ if } i \text{ eq } 0 \text{ then } [X]_{\langle link(i, u, stl^n(u)), i \rangle} \\ &\quad \text{else } [Y]_{\langle link(i, u, stl^n(u)), i-1 \rangle} \\ &= \lambda i \in \omega . \text{ if } i \text{ eq } 0 \text{ then } [g^n call_i X]_{\langle u, i \rangle} \\ &\quad \text{else } [g^n call_i Y]_{\langle u, i-1 \rangle} \\ &= [(g^n call_i X) fby (g^n call_i Y)]_{\langle u, i \rangle} \end{aligned}$$

■

(DR3') Prove that  $\gamma^n (X fby Y) = (\gamma^n X) fby (\gamma^n Y)$

$$[\gamma^n (X fby Y)]_{\langle u, i \rangle} = [(X fby Y)]_{\langle stl^n(u), i \rangle}$$

$= \lambda i \in \omega . \text{ if } i \text{ eq } 0 \text{ then } [X]_{\langle st]^n(u), i \rangle}$

$\text{ else } [Y]_{\langle st]^n(u), i-1 \rangle}$

$= \lambda i \in \omega . \text{ if } i \text{ eq } 0 \text{ then } [\gamma^n X]_{\langle u, i \rangle}$

$\text{ else } [\gamma^n Y]_{\langle u, i-1 \rangle}$

$= [(\gamma^n X) \text{ fby } (\gamma^n Y)]_{\langle u, i \rangle}$

■



#### 6.4- Evaluating Fluid:

Like any other member of our target language *DE*, there are two methods for evaluating Fluid-programs. The **reduction** method which is based on the rewrite rules of the algebra *Flu*; and the **eduction** method which is a demand driven tagged evaluation and is based on the definition of the algebra *Flu*. Of course, any of these methods can be implemented on a conventional machine. However, the first lends itself to reduction machines, such as ALICE [DaRe81] and Berkling's [Ber76]; while the other to demand driven tagged dataflow machines such as the Tagged Eduction Engine [As84].

The reader is reminded that the reduction method is probably impractical, and our main interest in this thesis is exclusively with the eduction method. In the next section however, we present the reduction method for evaluating Fluid programs to emphasize the difference between such a method and eduction.

##### 6.4.1- The Reduction Method:

Given an extensional algebra *A*, the rewrite rules for the algebra *Flu*, described in section 5.6, together with those of the extensional algebra *A* constitute a set of reduction rules for the equational language Fluid(*A*).

As described in section 5.1, reduction is a purely syntactic process. Given a program *P* in Fluid(*A*), an evaluation of *P* is a sequence of expressions,  $t_0, \dots, t_{n-1}$  where  $t_0$  is the expression **result**; and for every *i*,  $t_i$  is a consequence of *rewriting*  $t_{i-1}$  using

either a rewrite rule of the algebra *Flu*(*A*)

or a definition in the program *P*. We give here an example on evaluating Fluid using the rewrite rules of *Flu*. For more details on rewrite rules and reduction sequences the reader is referred to Chapter 5.

Example: (sieve of Eratosthenes)

Consider the following program in Luswim(N), which generates the infinite sequence of prime numbers in an ascending order.

```
sieve(n) where
  n = 2 fby n+1;
  sieve(X) = X fby sieve(notmultiple)
  where
    notmultiple = X wvr (X mod first X ne 0);
  end;
end
```

The corresponding Fluid(N)-program is:

```
result = call0 sieve          ...(1)

n = 2 fby n+1                  ...(2)

sieve = X fby call1 sieve      ...(3)

notmultiple = X wvr (X mod first X ne 0)    ...(4)

X = act(n,notmultiple)         ...(5)
```

The first expression in the reduction sequence is **result**. Notice that, for the ease of reference, we have numbered the equations of the Fluid-program.

```
result = call0 sieve
        = call0 (X fby call1 sieve)  (3 & subs'n)
        = call0 X fby (call0 call1 sieve)  (DR)
        = call0 (act(n,notmultiple)) fby (call0 call1 sieve)  (5, subs'n)
        = n fby (call0 call1 sieve)  (by Flo 1)  ...(*1)

call0 call1 sieve = call0 call1 (X fby call1 sieve)  (by 3 & subs'n)

        = (call0 call1 X) fby (call0 call1 call1sieve)  (DR)
        ...(*2)

call0 call1 X = call0 call1 (act (n,notmultiple))  (5 & subs'n)
        = call0 notmultiple  (Flo 1)
```

$$\begin{aligned}
&= \text{call}_0 (X \text{ wvr } (X \bmod \text{first } X \neq 0)) \quad (4 \text{ \& subs'n}) \\
&= \text{call}_0 X \text{ wvr } \text{call}_0 (X \bmod \text{first } X \neq 0) \quad (\text{by DR}) \\
&= (\text{call}_0 X) \text{ wvr } (\text{call}_0 X \bmod \text{call}_0 \text{first } X \neq 0) \quad (\text{by DR}) \\
&= n \text{ wvr } (n \bmod \text{first } n \neq 0) \\
&\quad (\text{by 5 \& Flo 1 \& call}_0 X = n)
\end{aligned}$$

Notice that, when  $n = \langle 2, 3, 4, 5, 6, 7, 8, 9, \dots \rangle$

$$\text{call}_0 X = n = \langle 2, 3, 4, 5, 6, 7, 8, 9, \dots \rangle$$

$$\begin{aligned}
\text{call}_0 \text{call}_1 X &= n \text{ wvr } (n \bmod \text{first } n \neq 0) \\
&= \langle 3, 5, 7, 9, 11, 13, \dots \rangle
\end{aligned}$$

If one continues the reduction process, then

$$\text{call}_0 \text{call}_1 \text{call}_1 X = \langle 5, 7, 11, 13, 17, 19, \dots \rangle.$$

The final resultant of the reduction process on the program will be the infinite expression

$$\begin{aligned}
&\text{call}_0 X \text{ fby } \text{call}_0 \text{call}_1 X \text{ fby } \text{call}_0 \text{call}_1 \text{call}_1 X \\
&\quad \text{fby } \text{call}_0 \text{call}_1 \text{call}_1 \text{call}_1 X \text{ fby } \dots \text{ fby } \text{call}_0 \text{call}_1^2 X \text{ fby } \dots
\end{aligned}$$

which equals the infinite expression

$$\begin{aligned}
&\langle 2, 3, 4, 5, 6, 7, \dots \rangle \text{ fby } \langle 3, 5, 7, 9, 11, 13, \dots \rangle \\
&\quad \text{fby } \langle 5, 7, 11, 13, 17, \dots \rangle \\
&\quad \text{fby } \langle 7, 11, 13, 17, \dots \rangle \text{ fby } \dots
\end{aligned}$$

which is equal to the infinite sequence

$$\langle 2, 3, 5, 7, 11, 13, \dots \rangle \quad \blacksquare$$

6.4.2- Evaluating Fluid by Eduction:

In section 5.3, we have described the evaluation of Florid by tagged eduction. For an algebra A, the language Florid(A) is  $DE(Fl_o(A))$  where the universe of  $Fl_o(A)$  is the set of lists with back pointers over the natural numbers. Thus a data token (a daton) in the eduction of Florid(A) is a tagged expression. The tag part of such a daton is an element in the universe of  $Fl_o$ ; i.e. it is a list with back pointer.

$Fl_o(A)$ -expression	tag: b-list
-----------------------	-------------

a data token in the eduction of Florid(A)

The language Fluid(A) is  $DE(Fl_u(A))$ , and the universe of  $Fl_u$  is the set  $bl(\omega) \times \omega$ . Each world in such a universe is a pair representing an element in  $bl(\omega)$  together with an element in  $\omega$ . Thus, in the eduction of Fluid, the tag in a data token should consist of two cells. These two cells are a pair representing an element in the universe  $bl(\omega) \times \omega$ . In other words, one cell for the place and another for the time. That is,

$Fl_o(A)$ -expression	tag	
	place $\in bl(\omega)$	time $\in \omega$

a data token in the eduction of Fluid(A)

The value of a Fluid(A)-program is the value of **result** at the b-list  $\Lambda$ , which is the infinite sequence

$$\lambda i \in \omega . [result]_{\langle \Lambda, i \rangle}$$

There are two methods to practically implement the eduction procedure in

evaluating **result**. The first is to start with  $i=0$ ; i.e evaluate  $[\text{result}]_{\langle \Lambda, 0 \rangle}$ , then increment  $i$ , and so on. Hence, producing the infinite sequence

$$\lambda i \in \omega . [\text{result}]_{\langle \Lambda, i \rangle}$$

This method can be implemented on a multiprocessor system, but it is more suitable for mono-processor machines. This is how the Warwick implementation of Plucid works [FMY83].

The second method is suitable for multi-processor systems. More than one demand for the value of **result** at different times are processed at the same time. That is, we can start the eduction process by demanding simultaneously

$$[\text{result}]_{\langle \Lambda, 0 \rangle} , [\text{result}]_{\langle \Lambda, 1 \rangle} , \dots , [\text{result}]_{\langle \Lambda, n \rangle}$$

where the number  $n$  here is dependent on the size and the capability of the hardware. Once, one of these value is computed, a demand for the  $j$ 'th value of **result**, for some  $j > n$  is created, and so on.

Another possibility is to implement an interactive system within which the machine asks for the instance of **result** required, given a natural number  $k$  the machine, then, evaluates

$$[\text{result}]_{\langle \Lambda, k \rangle}$$

Choosing a particular strategy, from the above, is dependent on many considerations; such as the hardware available, the source language being implemented and the purpose of the system. Our interest in this thesis is not committed to any particular evaluation strategy or to any particular hardware arrangement; we want to show that given a Fluid-program together with a natural number, say  $j$ , we can educe (evaluate) the  $j$ 'th value of such a program using the definition of the algebra *Flu*.

Let us consider, for example, the program for generating the infinite sequence of prime numbers given in section 6.4.1. That is, the  $\text{Luswim}(N)$



program

```
sieve(n) where
  n = 2 fby n+1;
  sieve(X) = X fby sieve(notmultiple)
    where
      notmultiple = X wvr (X mod first X ne 0);
    end;
end
```

The equivalent Fluid(N)-program is

```
result = call0 sieve          ...(1)
n = 2 fby n+1                  ...(2)
sieve = X fby call1 sieve      ...(3)
notmultiple = X wvr (X mod first X ne 0)  ...(4)
X = act(n,notmultiple)         ...(5)
```

Assume, also, that we want to evaluate the second element in the value of the program; that is, the second element in the infinite sequence of prime numbers. Thus, we want the value of

$[result]_{\langle \Lambda, 1 \rangle}$

Using the meta-function Dem, described in section 5.4, the evaluation can be described as follows:

```
Dem (result , <Λ,1>)
= Dem (call0 sieve , <Λ,1>) (1,subs'n)
= Dem (sieve , <<0,Λ,Λ>, 1>) (def'n of call)
= Dem (sieve , <α, 1>) where α = <0,Λ,Λ>
= Dem (X fby call1 sieve , <α, 1>) (3,subs'n)
= Dem (call1 sieve , <α, 0>) (def'n of fby)
= Dem ( sieve , <<1,α,α>, 0>) (def'n of call)
= Dem ( sieve , <β, 0>) where β = <1,α,α>
```

$$\begin{aligned}
&= \text{Dem} (X \text{ fby } \text{call}_1 \text{ sieve}, < \beta, 0 >) \quad (3, \text{subs}'n) \\
&= \text{Dem} (X, < \beta, 0 >) \quad (\text{def}'n \text{ of fby}) \\
&= \text{Dem} (\text{act}(n, \text{notmultiple}), < \beta, 0 >) \quad (5, \text{subs}'n) \\
&= \text{Dem} (\text{notmultiple}, < \alpha, 0 >) \quad (\text{def}'n \text{ of act}) \\
&= \text{Dem} (X \text{ wvr } (X \text{ mod first } X \text{ ne } 0), < \alpha, 0 >) \\
&\quad (4, \text{subs}'n) \quad \dots (*1)
\end{aligned}$$

Notice that, for any expressions  $x, y$

$$x \text{ wvr } y = \lambda i \in \omega . \text{ if } y_i \text{ then } x_i \text{ else } \perp$$

Thus we start searching for  $k \in \omega$  such that

$$\text{Dem} (X \text{ mod first } X \text{ ne } 0, < \alpha, k >) \text{ is true} \quad \dots (*2)$$

For  $k=0$  :

$$\begin{aligned}
&\text{Dem} (X \text{ mod first } X \text{ ne } 0, < \alpha, 0 >) = \\
&\quad \text{Dem}(X, < \alpha, 0 >) \text{ mod } \text{Dem}(\text{first } X, < \alpha, 0 >) \text{ ne } \text{Dem}(0, < \alpha, 0 >) \\
&= \text{Dem} (X, < \alpha, 0 >) \text{ mod } \text{Dem} (\text{first } X, < \alpha, 0 >) \text{ ne } 0 \\
&\quad \dots (*3)
\end{aligned}$$

$$\begin{aligned}
&\text{Dem}(X, < \alpha, 0 >) = \text{Dem}(\text{act}(n, \text{notmultiple}), < \alpha, 0 >) \quad (5, \text{subs}'n) \\
&= \text{Dem}(n, < \Lambda, 0 >) \quad (\text{def}'n \text{ of act}) \\
&= \text{Dem}(2 \text{ fby } n+1, < \Lambda, 0 >) \quad (2, \text{subs}'n) \\
&= 2 \quad (\text{def}'n \text{ of fby})
\end{aligned}$$

Similarly, the reader can verify that

$$\text{Dem} (\text{first } X, < \alpha, 0 >) = 2$$

Thus, (\*3) above is equal to false.

For  $k=1$  :

$$\begin{aligned}
&\text{Dem} (X \text{ mod first } X \text{ ne } 0, < \alpha, 1 >) = \\
&\quad \text{Dem}(X, < \alpha, 1 >) \text{ mod } \text{Dem}(\text{first } X, < \alpha, 1 >) \text{ ne } \text{Dem}(0, < \alpha, 1 >) \\
&= \text{Dem}(X, < \alpha, 1 >) \text{ mod } \text{Dem}(\text{first } X, < \alpha, 1 >) \text{ ne } 0 \quad \dots (*4)
\end{aligned}$$

$$\begin{aligned}
& \text{Dem}(X, \langle \alpha, 1 \rangle) = \text{Dem}(\text{act}(n, \text{notmultiple}), \langle \alpha, 1 \rangle) \quad (\text{subs'n}) \\
& = \text{Dem}(n, \langle \Lambda, 1 \rangle) \quad (\text{def'n of act and that of } \alpha) \\
& = \text{Dem}(2 \text{ fby } n+1, \langle \Lambda, 1 \rangle) \quad (2, \text{subs'n}) \\
& = \text{Dem}(n+1, \langle \Lambda, 0 \rangle) \quad (\text{def'n of fby}) \\
& = \text{Dem}(n, \langle \Lambda, 0 \rangle) + \text{Dem}(1, \langle \Lambda, 0 \rangle) \\
& = \text{Dem}(2 \text{ fby } n+1, \langle \Lambda, 0 \rangle) + 1 \\
& \quad (2, \text{subs'n } \& 1 \text{ is a constant symbol}) \\
& = 2 + 1 = 3 \quad (\text{def'n of fby}) \quad \dots(*5)
\end{aligned}$$

Similarly, the reader can verify that  $\text{Dem}(\text{first } X, \langle \alpha, 0 \rangle) = 2$

Thus the value of (\*4) above is true, and hence  $k$  in (\*2) above is 1

Therefore,  $\text{Dem}(\text{result}, \langle \Lambda, 1 \rangle) =$

$$\text{Dem}(X \text{ wvr } (X \text{ mod first } X \text{ ne } 0), \langle \alpha, 0 \rangle) \quad (\text{by } *1)$$

and  $\text{Dem}(X \text{ mod first } X \text{ ne } 0, \langle \alpha, k \rangle)$  is true when  $k$  is 1

(by \*4 & \*5)

thus

$$\begin{aligned}
& \text{Dem}(\text{result}, \langle \Lambda, 1 \rangle) = \text{Dem}(X, \langle \alpha, 1 \rangle) \\
& = \text{Dem}(\text{act}(n, \text{notmultiple}), \langle \alpha, 1 \rangle) \\
& = \text{Dem}(n, \langle \Lambda, 1 \rangle) \\
& = 3
\end{aligned}$$

# Chapter 7

## Conclusions and Further work

### 7.0- Conclusion:

The subject of this dissertation is to lay down a general approach for implementing programming languages. Our model can be used to implement conventional languages; however, our interest is mainly with functional languages in general and dataflow languages (such as Lucid) in particular.

Roughly speaking, given any functional language the first step in the process of implementing such a language, using our approach, is to construct the target intensional algebra for the language. The source language is then compiled into an equational nullary order language (such as *DE*) whose terms (or expressions) are over the terms of the target algebra. In order to construct the appropriate target algebra, we have to look for a mathematical structure to be a universe for such an algebra; furthermore, we need the algebra to comprise a collection of intensional operators over such a universe. Informally speaking, the complexity of such an algebra is dependent on the language to be compiled.

For example, when compiling the language *Iwade* (a first order variant of *Iswim* which does not allow occurrences of globals in function definitions), we defined the algebra *FUN* whose universe is the set of function calls. A function call is represented as a list of natural numbers; the head of the list represents the call of the present function (the one being invoked) and the tail represents the list of function calls which lead to the present one. The intensional

operators used in *FUN* are two; *call* which stacks the present call on top of the list of calls and *act* which bundles the actual parameters associated with a certain formal parameter according to their occurrences in the program. When implementing the language *lswum* (lswade with occurrences of global variable symbols), we defined the universe to be the set of lists with back pointers where a back pointer points at the place in the universe where the function (being called) is defined. The intensional algebra *Flo* (the target algebra for implementing *lswum*) comprises intensional operators for calling functions, both local and global ones, (such as *call*, *gcall*, *ggcall*, and so on). It also comprises operators for calling nullary globals (such as  $\gamma$ ,  $\gamma\gamma$ , etc).

This procedure of compiling, we believe, can be extended to handle more complex languages. We have demonstrated in Chapter 6 the procedure of compiling the language *luswim* by defining the algebra *Flu* of place and time.

For each intensional algebra, we defined a collection of rewrite rules, and formally proved their correctness (based upon the algebra itself). Our target language *DE* is an equational language over such intensional algebras; thus the rewrite rules of the algebra induces a rewrite system (a reduction schema) on the expressions of the target language. Such a schema can be used as an evaluation technique for those expressions (the reduction method).

Another model for evaluating the expressions of *DE* is the "eduction" model. This model of evaluation, proposed as a main theme in our thesis, is demand-driven tagged dataflow. It is a pure form of tagged lazy evaluation. In fact, a better title to this dissertation would be "an intensional approach to tagged lazy evaluation" or "the theory of tagged eduction".

Nevertheless, there are some points which suggested themselves while carrying out this research. Some of them are extensions to what we have proposed and others are open problems we would like to pursue in our future



work. In the rest of this chapter, we cast some light on these points and (where possible) outline a general procedure for tackling them.

### 7.1- Implementing Currenting in Lucid:

As a direct consequence of our work, and as our interest is with Lucid, the first main extension is to implement currenting (or freezing) in Lucid; hence an implementation of full Lucid.

We shall first examine the process of compiling Lucid without user-defined functions. That is, we want an algebra which preserves the meanings of the Lucid operators and, at the same time, facilitates nested iteration. Such an algebra (the target of Lucid without function definitions) will be joined with the algebra *Flu* described in section 4.7. The join will be along the same lines as the construction of *Flu* described in section 6.1.

Consider the following pLucid program for generating the sequence of powers  $X^M$ :

```

pow where
  X = 1 fby X+1 ;
  M = 0 fby M+2;
  pow = res where
    x is current X; // freeze the current value of X
    m is current M; // freeze the current value of M
    I = 0 fby I+1;
    P = 1 fby P*x;
    res = P asa I eq m;
  end;
end

```

The value of  $P$ , in the inner clause, at any time  $t$  equals the value of the expression  $1 \text{ fby } P * x$  at  $t$ . Ideally, we want the operations of the new algebra to preserve the meaning of **fby** and **\***. That is, we want **\*** to be interpreted pointwise. Thus, the value of  $P$  at  $t$  (where  $t \neq 1$ ) equals the value of the expression  $P_{t-1} * x_{t-1}$ . The value of  $P_{t-1}$  can be recursively computed using the same equation. However, the only piece of information we know, from the

assertion

**x is current X**

is that, for every instances in time  $i$  and  $j$  ( $x_i = x_j$ ).

This information is not enough to compute the value of  $x$  at a certain instance in time. We need to know the global time at which  $X$  was frozen. If we let  $t'$  be the frozen global time, then the value of  $x_t$  should be equal to  $X_{t'}$ . Thus, while the universe of  $\mathcal{L}\mathcal{U}$  is the set of natural numbers  $\omega$ , the universe of the target algebra for implementing nested iteration should be the set of sequence of natural numbers  ${}^\omega\omega$ . For a sequence  $q$  in such a universe, the first element (the head) is the present (local) time while the tail is the sequence of the so far frozen global times.

For example, the value of  $x$  at time  $\langle 3, 2 \rangle$  is the value of **is current X** at time  $\langle 3, 2 \rangle$ , which should be the value of  $X$  at the frozen global time; that is, at time 2. Thus, the value of  $x$  at  $\langle 3, 2 \rangle$  equals the value of  $X$  at 2 which is the value 3. Consequently, the value of  $P$  at  $\langle 3, 2 \rangle$  is the value of

**1 fby  $P * x$  at  $\langle 3, 2 \rangle$**

which is the infinite sequence

$\langle 1, 3, 9, 27, 81, \dots \rangle$

The reader should notice that, we interpret the symbol  $*$  pointwise over the universe, and hence

$$(P * x)_{\langle 3, 2 \rangle} = P_{\langle 3, 2 \rangle} * x_{\langle 3, 2 \rangle}$$

Furthermore, the first element of a time sequence denotes the present local time and, only it, is the subject of any current computation or evaluation. Also, if we think of **x is current X** as a definition stating that  $x$  is the frozen value of the global  $X$ , and denote it by  $x = \text{Fr}(X)$ , where  $\text{Fr}$  is an intensional operator symbol; then the value of

$x_{\langle 3, 2 \rangle}$  is equal to the value of  $\text{Fr}(X)_{\langle 3, 2 \rangle}$

which should be equal to the value of  $X$  at  $\langle 2 \rangle$ .

Informally speaking, we define the universe of the target algebra to be the set of infinite sequence of non-negative integers  ${}^\omega\omega$ . On this set, we define the following three operations which correspond to the usual functions defined on lists. The operation *glotime* which returns the tail of the sequence (the history of global time), the operation *loctime* which returns the head of the sequence (which denotes the present or local time), and the sequence constructor *seq* which takes a natural number  $n$  and a sequence  $q$  and returns the sequence whose head (or loctime) is  $n$  and whose tail (or glotime) is  $q$ .

*Definition:*

Define  $\Omega$  to be the set  ${}^\omega\omega$

i.e. the set of finite and infinite sequences of natural numbers.

Define the operations loctime, glotime, and seq on the set

$\Omega \cup \omega$  as follows:

a- For every  $t \in \Omega$ ,  $\text{loctime}(t) = t_0$

i.e. it returns the first element of the sequence.

b- For every  $t \in \Omega$ ,  $\text{glotime}(t) = \lambda n \in \omega [t(n+1)]$

i.e. it returns the tail of the sequence

c- For every  $n \in \omega$  and for every  $t \in \Omega$

$\text{seq}(n, t) = \lambda m [ \text{if } m \text{ eq } 0 \text{ then } n \text{ else } t_{m-1}]$

**Examples :**

$\text{loctime}(\langle x_0, x_1, x_2, \dots, x_n, \dots \rangle) = x_0$

$\text{glotime}(\langle x_0, x_1, x_2, \dots, x_n, \dots \rangle) = \langle x_1, x_2, \dots, x_{n-1}, \dots \rangle$

$\text{seq}(n, \langle x_0, x_1, x_2, \dots, x_n, \dots \rangle) = \langle n, x_0, x_1, x_2, \dots, x_n, \dots \rangle$

To implement nested iteration in Lucid, we introduce the family of intensional algebras  $LL\mathcal{A}$ . The universe of  $LL\mathcal{A}(A)$ , for a data algebra  $A$ , is the set of infinite sequences  $\Omega (=^\omega\omega)$ . The signature of  $LL\mathcal{A}(A)$  is the union of the signature of  $A$ , the Lucid operator symbols and the intensional operator symbols **Fr** and **nest**. The Lucid expression

*var is current expr*

is translated into  $var = \text{Fr}(\text{expr})$

Given any sequence  $t$  in the universe  $\Omega (=^\omega\omega)$

and any  $LL\mathcal{A}(A)$ -expression  $\varepsilon$

$$LL\mathcal{A}(A)(\text{Fr}(\varepsilon))_t = LL\mathcal{A}(A)(\varepsilon)_{\text{glotime}(t)}$$

The symbol **nest** is used for compiling the result of the currented where clause. For example, the definition

**X = Z + W - Y where**

**Y is current H+3-K;**

**W = ...;**

**Z = ...;**

**end;**

is compiled into

**X = nest(Z + W - Y)** together with the definitions resulted from compiling the definitions inside the where clause including of course the definition

$$Y = \text{Fr}(H+3-K)$$

For any  $LL\mathcal{A}(A)$  expression  $\varepsilon$ , The meaning of **nest**( $\varepsilon$ ) given by the algebra  $LL\mathcal{A}(A)$  is defined as follows:

For any  $t \in {}^\omega\omega$

$$LL\mathcal{A}(A)(\text{nest}(\varepsilon))_t = LL\mathcal{A}(A)(\varepsilon)_{\text{seq}(0,t)}$$

Thus while  $LL\mathcal{A}(A)(\text{nest})$  starts a new subcomputation by pushing an instance in time at the beginning of the time sequence, the operator  $LL\mathcal{A}(A)(\text{Fr})$

pops out the present time and returns to the latest (frozen) global time.

We give here the formal definition of the family of intensional algebras **LL $\mathcal{A}$** :

**The Definition of **LL $\mathcal{A}$**  :**

Let  $A (= \langle F, D \rangle)$  be an extensional  $\Sigma$ -algebra.

The intensional  $\Sigma$ -algebra **LL $\mathcal{A}$** ( $A$ ) is the triple

$\langle \Omega, F', D \rangle$  where

a: the universe  $\Omega$  is the set  ${}^\omega\omega$

b.1: the intensional interpretation function  $F'$  extends  $F$  pointwise,

i.e. for every  $n$ -ary constant symbol  $\psi$  in  $\Sigma$ ,

for every  $n$  terms  $a_0, \dots, a_{n-1}$  in **LL $\mathcal{A}$** ( $A$ )

and for every  $t \in \Omega$

$$(F'(\psi)(a_0, \dots, a_{n-1}))_t = F(\psi)(F'(a_0)_t, \dots, F'(a_{n-1})_t)$$

b.2: the function  $F'$  assigns meaning to the constant symbols

**first, next, fby, wvr, asa, upon, attime**

as follows :

For every  $X, Y, Z$  in  ${}^nD$ , and for every sequence  $t \in \Omega$

$$F'(\mathbf{first})(X)_t = X_{\text{seq}(0, \text{glotime}(t))}$$

$$F'(\mathbf{next})(X)_t = X_{\text{seq}(\text{loctime}(t)+1, \text{glotime}(t))}$$

$$F'(\mathbf{fby})(X, Y)_t = \text{if } (\text{loctime}(t) \text{ eq } 0) \text{ then } X_{\text{seq}(0, \text{glotime}(t))} \\ \text{else } Y_{\text{seq}(\text{loctime}(t)-1, \text{glotime}(t))}$$

$$F'(\mathbf{wvr})(X, Z)_t = \text{if } (Z_t \text{ eq true}) \text{ then } X_t \\ \text{else } \perp$$



$F'(\text{asa})(X,Z)_t = \text{if } \exists j \leq \text{loctime}(t) \text{ such that}$

$(Z_{\text{seq}(j, \text{glotime}(t))} \text{ eq true}) \text{ and } (k \leq j \rightarrow \text{not } (Z_{\text{seq}(k, \text{glotime}(t))}))$

$\text{then } X_{\text{seq}(j, \text{glotime}(t))}$

$\text{else } \perp$

$F'(\text{upon})(X,Z)_t = X_{\text{seq}(n, \text{glotime}(t))}$

where  $n = \text{Card}\{j \leq \text{loctime}(t) : Z_{\text{seq}(j, \text{glotime}(t))} \text{ eq true}\}$

and  $\text{Card}$  is the cardinality function on sets.

$F'(\text{attime})(X,Z)_t = X_{Z(\text{seq}(Z, \text{glotime}(t)))}$

b.3: The function  $F'$  assigns meaning to the symbols

$\text{nest}$ , and  $\text{Fr}$  as follows:

For every  $X$  in  ${}^{\Omega}D$ , and for every  $t$  in  $\Omega$

$F'(\text{nest}(X))_t = X_{\text{seq}(0, t)}$

$F'(\text{Fr}(X))_t = X_{\text{glotime}(t)}$

As a target algebra for implementing full Lucid (Lucid with nested iteration and user-defined functions) we construct the algebra  $FLL\mathcal{U}$ . For a data algebra  $A$ ,  $FLL\mathcal{U}(A)$  is the join of  $LL\mathcal{U}(A)$  together with  $FLo(A)$ . The construction of the join of these two algebras is similar to the construction of  $Fl\mathcal{U}$  described in section 6.1.

Thus, the universe of  $FLL\mathcal{U}(A)$  is the product of the universe  $\Omega (= {}^{\omega}\omega)$  of  $LL\mathcal{U}(A)$  together with the universe  $\text{bl}(\omega)$  of  $FLo(A)$ . That is, it is the set  $\Omega \times \text{bl}(\omega)$ . Furthermore,  $FLL\mathcal{U}(A)$  extends the operations of the algebras  $LL\mathcal{U}(A)$  and  $FLo(A)$  pointwise upon such a universe. For example,

for any  $FLL\mathcal{U}(A)$ -expression  $X$ , and  $\langle t, p \rangle$  in  $\Omega \times \text{bl}(\omega)$

$(FLL\mathcal{U}(A)(\text{first}(X)))_{\langle t, p \rangle} = (FLL\mathcal{U}(A)(X))_{\langle \text{seq}(0, \text{glotime}(t)), p \rangle}$

$L\mathcal{L}\mathcal{U}(A)$  and  $F\mathcal{L}\mathcal{O}(A)$  pointwise upon such a universe. For example,

for any  $FLL\mathcal{U}(A)$ -expression  $X$ , and  $\langle t, p \rangle$  in  $\Omega \times bl(\omega)$

$$(FLL\mathcal{U}(A)(\text{first}(X)))_{\langle t, p \rangle} = (FLL\mathcal{U}(A)(X))_{\langle seq(0, glotime(t)), p \rangle}$$

$$(FLL\mathcal{U}(A)(\text{next}(X)))_{\langle t, p \rangle} = (FLL\mathcal{U}(A)(X))_{\langle seq(loctime(t)+1, glotime(t)), p \rangle}$$

$$(FLL\mathcal{U}(A)(\text{fby}(X, Y)))_{\langle t, p \rangle} =$$

$$\text{if } (loctime(t) \text{ eq } 0) \text{ then } (FLL\mathcal{U}(A)(X))_{\langle seq(0, glotime(t)), p \rangle}$$

$$\text{else } (FLL\mathcal{U}(A)(Y))_{\langle seq(loctime(t)-1, glotime(t)), p \rangle}$$

for any  $i \in \omega$

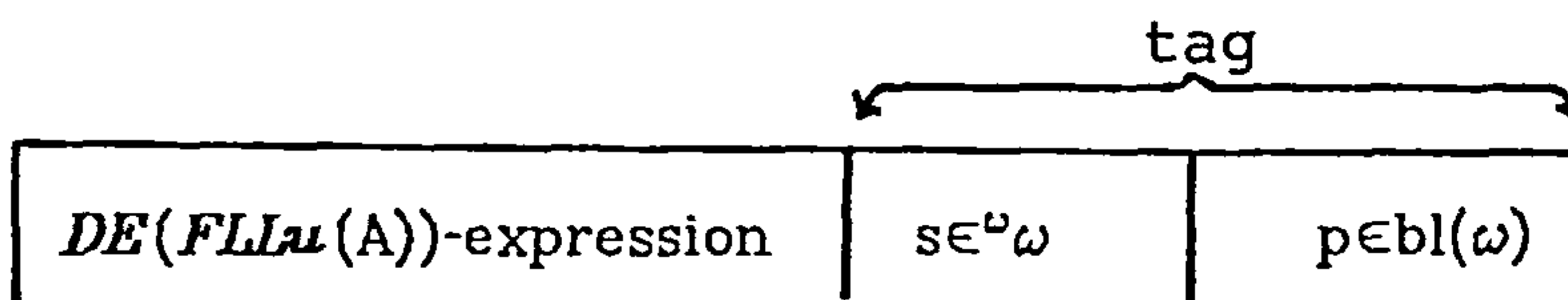
$$(FLL\mathcal{U}(A)(\text{call}_i(X)))_{\langle t, p \rangle} = (FLL\mathcal{U}(A)(X))_{\langle t, \text{link}(i, p, p) \rangle}$$

for any  $n$  expressions  $x_0, \dots, x_{n-1}$  in  $FLL\mathcal{U}(A)$

$$(FLL\mathcal{U}(A)(\text{act}(x_0, x_1, \dots, x_{n-1})))_{\langle t, p \rangle} = (FLL\mathcal{U}(A)(x_{hd(p)}))_{\langle t, dl(p) \rangle}$$

and so on.

Therefore, in evaluating Lucid expressions by tagged eduction, the expressions should be tagged with two cells; one contains a sequence of natural numbers (representing instances in time) and the other cell contains a list with back pointer (representing a place in the tree of function calls). That is, for an extensional data algebra  $A$ , a tagged expression in the eductive evaluation of  $DE(FLL\mathcal{U}(A))$ , which is the target language for Lucid (with user-defined functions and nested iteration) can be illustrated in the following diagram:



## 7.2- The Correctness of the Compilation Algorithm:

We have introduced, in sections 3.6, an algorithm for compiling the family of functional languages *Iwade* into the family of intensional equational nullary order languages *DE(FUN)*. In this section, we suggest general lines for proving the correctness of this algorithm.

Let  $A$  be an extensional algebra of data types,  
and let  $P$  be a program in the language *Iwade*( $A$ ),  
then  $\text{Trans}(P)$  is a program in the language *DE(FUN)*( $A$ ).

Informally speaking, we have to prove that the meaning of a program  $P$  in *Iwade*( $A$ ) is equal to the meaning of the program  $\text{Trans}(P)$  in *DE(FUN)*( $A$ ). However, there is a slight falsity in this argument because the meaning of an expression in *Iwade* is an extensional object, while the meaning of a program in *DE* is intensional. Therefore, if  $D$  is the domain of the algebra  $A$  then the meaning of a program  $P$  in *Iwade*( $A$ ) is a function on the objects of  $D$ ; that is, the meaning of  $P$  is of functionality  $[D^n \rightarrow D]$  for some natural number  $n$ . However, the value of  $\text{Trans}(P)$  is a function on the intensional domain  ${}^L D$  where  $L$  is the set of lists of natural numbers which is the universe of *FUN*( $A$ ). That is, the meaning of  $\text{Trans}(P)$  is of functionality  $[({}^L D)^n \rightarrow {}^L D]$ .

Therefore, to justify the correctness of the compilation algorithm, we have to prove that the value of the program  $P$  in *Iwade*( $A$ ) is equal to the value of  $\text{Trans}(P)$  in *DE(FUN)*( $A$ ) at the empty list  $\text{nil}$  in  $L$  which is the world of commencing the evaluation.

Another major point of concern when proving the correctness is that an environment for *Iwade*( $A$ ) should be of functionality  $[V \rightarrow \bigcup_{n \in \omega} [D^n \rightarrow D]]$ , where  $V$  is the set of variable symbols; however an environment for *DE(FUN)*( $A$ ) is of functionality  $[V \rightarrow {}^L D]$ . The reader should notice here that occurrences of

variable symbols in  $DE(FUN(A))$  are all nullaries. Furthermore, when comparing the meaning of  $Iwade(A)(P)$  with the meaning of  $DE(FUN(A))(Trans(P))$ , the  $Iwade(A)$ -environment (in which  $P$  is evaluated) should be *equivalent* to the  $DE(FUN(A))$ -environment. *Equivalent* here means that they should agree on the meanings assigned to the variable symbols.

If we denote the meaning of  $P$  in an environment  $\varepsilon$  by  $Iwade(A)(P)_\varepsilon$ , then we conjecture the following theorem for the correctness of the compilation algorithm:

**Theorem:**

Given an extensional algebra  $A$ , and a program  $P$  in  $Iwade(A)$ ,

$$Iwade(A)(P)_\varepsilon = ( DE(FUN(A))(Trans(P)) )_{\varepsilon'} (nil)$$

where  $\varepsilon'$  is defined as follows:

$$\text{for every variable symbol } v, \quad \varepsilon'(v) = \lambda v. [\lambda l \in L. \varepsilon(v)]$$

### 7.3- Other Areas:

Probably, the first question which is asked when either implementing or designing a functional language is whether the language supports (or allows) higher order functions. Lucid is a first order iterative language and does not allow (at least at this stage) higher order functions in its programs. We believe, however, that extending Lucid to allow such functions is possible. We believe also that implementing higher order functions using intensional logic (our approach) is possible. We feel that an intensional algebra whose universe is the set of trees of trees (jungles) of natural numbers can be constructed as a target algebra for this purpose. However, we leave the problem open for research.

Another area for future investigation is machine architecture. That is,

building a demand-driven tagged dataflow machine whose low level code is a member of the family of intensional languages *DE*. A machine based on these ideas, however, has been proposed by E.Ashcroft at SRI<sup>†</sup>. It is called "A Tagged Education Engine" [Ash84].

---

<sup>†</sup> : Stanford Research Institute International, California, USA.



## Appendix A :

We give here a BNF formalism for the pLucid syntax (from [FMY83]).

where

`::=` is read as `<meta variable>` is defined as `<meta variable>`,

`|` is read as `<meta variable>` or `<meta variable>` ,

`{ }` denotes possible repetition zero or more times  
of the enclosed construct .

`<program> ::= <expression>`

`<expression> ::= <constant>`

`| <identifier>`

`| error`

`| eod`

`| <prefix operator> <expression>`

`| <expression> <infix operator> <expression>`

`| filter (<expression>,<expression>,<expression>)`

`| substr (<expression>,<expression>,<expression>)`

`| length <expression>`

`| arg <expression>`

`| <list expression>`

`| <if expression>`

`| <case expression>`

`| <cond expression>`

`| <function call>`

`| <where clause>`

`<constant> ::= <numeric constant>`

`| <word constant>`

`| <string constant>`

`| <list constant>`

`<numeric constant> ::= <integer constant>`

`| <real constant>`

`<integer constant> ::= <digit> { <digit> }`

`| <n-sign> <integer constant>`

`<real constant> ::= <integer constant> . { <digit> }`

`<n-sign> ::= ~`

`<word constant> ::= <quote> <word constant less the quotes> <quote>`

`<word constant less the quote> ::= <letter> { <alphanumeric> }`

`| <sign> {<sign>}`

`| <bracket>`

`| <period>`

`| <separator>`

`| <quote>`

<sign> ::= + | - | \* | | & | = | < | > | : | # | ^  
 <quote> ::= "  
 <bracket> ::= ( | ) | [ % | % ] | ( % | % )  
 <period> ::= .  
 <separator> ::= , | ;

<string constant> ::= '{<character>}'

<character> ::= <Any ASCII character except the closing single quote ' >

<list constant> ::= nil | []  
 | [ {<list constant element>} ]

<list constant element> ::= <numeric constant>  
 | <word constant less the quotes>  
 | <string constant>  
 | <list constant>

<alphanumeric> ::= <digit> | <letter>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M  
 | N | O | P | Q | R | S | T | U | V | W | X | Y | Z  
 | a | b | c | d | e | f | g | h | i | j | k | l | m  
 | n | o | p | q | r | s | t | u | v | w | x | y | z

<identifier> ::= <letter> { <alpahnumeric> }

<prefix operator> ::= <p-numeric operator>  
 | <p-word operator>  
 | <p-string operator>  
 | <p-list operator>  
 | <p-lucid operator>  
 | <p-special operator>

<p-numeric operator> ::= sin | cos | tan | log | isnumber | log10 | abs | sqrt

<p-word operator> ::= isword | not | mkstring

<p-string operator> ::= isstring | mkword

<p-list operator> ::= hd | tl | isatom | isnull

<p-lucid operator> ::= first | next

<p-special operator> ::= iseod | iserror

<infix operator> ::= <i-numeric operator>  
 | <i-word operator>  
 | <i-string operator>  
 | <i-list operator>  
 | <i-lucid operator>

<i-numeric operator> ::= + | - | \* | \*\* | div | mod | /  
 eq | ne | <= | < | > | >=

<i-word operator> ::= and | or | eq | ne

<i-string operator> ::= ^ | eq | ne

<i-list operator> ::= <> | :: | eq | ne

<i-lucid operator> ::= fby | whenever | wvr | upon | asa

<list expression> ::= [%%] | [% {<expressions list>} %]

<expressions list> ::= <expression item>  
| <expression item> , {<expressions list>}

<expression item> ::= <expression> | <list expression>

<if expression> ::= if <expression> then <expression> <endif>

<endif> ::= else <expression> fi  
| elsif <expression> then <expression> <endif>

<case expression> ::= case <expression> of <casebody> end

<cond expression> ::= cond <casebody> end

<casebody> ::= {<expression> : <expression> ;} <defaultcase>

<defaultcase> ::= default : <expression>;

<function call> ::= <identifier> ( <actuals list> )

<actuals list> ::= <expression> | <expression> , <actuals list>

<where clause> ::= <expression> where <body> end

<body> ::= <declarations list> <definitions list>

<declarations list> ::= { <current declaration> ; }

<current declaration> ::= <identifier> is current <expression>

<definitions list> ::= { <definition> ; }

<definition> ::= <simple definition> | <function definition>

<simple definition> ::= <identifier> = <expression>

<function definition> ::= <identifier> ( <formals list> ) = <expression>

<formals list> ::= <identifier> | <identifier> , <formals list>

## References:

- [Abr82] "SECD-M: a Virtual Machine for Applicative Multiprogramming"  
S. Abramsky, CSL report, Queen Mary College, Nov. 1982.
- [Ack78] "Preliminary Data Flow Language", W.B. Ackerman,  
CSG 36, Laboratory for Computer Science, MIT 1978.
- [Ack79] "Data Flow Languages", W. Ackerman,  
Proc. of National Computer Conference 1979,  
Pages 1087-1098
- [ArGo77] "Some Relationships Between Asynchronous Interpreters of a  
Dataflow language", Arvind, and K.P.Gostelow. in Formal  
Description of Programming Concepts,  
North Holland Pub. Co., New York, 1977.
- [ArGo78] "Dataflow Computer Architecture: Research and Goals",  
Arvind, and K.P.Gostelow, Tech. Rep #113, Dept of Information  
and Computer Science, Univ. of Calif, Irvine 1978.
- [ArKa81] "A Multiple Processor Dataflow Machine that Supports  
Generalized Procedures", Arvind and V.Kathail,  
Proc. of the 8th Annual Symp. on Computer Architecture,  
Comp. Architecture, May, 1981.
- [Ash84] "A Tagged Execution Engine", E. Ashcroft  
SRI International (Draft) 1984.  
(personal communication)
- [AsWa76] "Lucid- A Formal System for Writing and Proving Programs",  
E.Ashcroft and W. Wadge, SIAM J. of Computing, 5,  
No 3, 1976.
- [AsWa77a] "Lucid, a non Procedural Language with Iteration",  
E.Ashcroft and W.Wadge, CACM vol 20, No 7, 1977.
- [AsWa77b] "Scope Structures and Defined Functions in Lucid",  
E.Ashcroft and W.Wadge,  
Theory of Computation Report No 21, Univ. of Warwick, 1977.
- [AsWa79] "A Logical Programming Language", E.Ashcroft and W.Wadge,  
Report No. CS-79-20, Univ. of Waterloo, 1979.
- [AsWa80] "Structured Lucid", E.Ashcroft and W.Wadge, CS Report 33,  
Univ. of Warwick March 1980.
- [BCP71] "Programming in POP-2", R.Burstall, J.Collins, and  
R. Popplestone, Edinburgh Univ. Press 1971.
- [BMS80] "HOPE: An Experimental Applicative Language", R. Burstall,  
D.McQueen, and D. Sannella, Proc. LISP Conf, Stanford 1980.



- [Bac78] "Can Programming Be Liberated from the von Neumann style:  
a functional language and its algebra of programs"  
Turing Lecture, CACM Aug 1978.
- [Bac81] "Is Computer Science Based on the Wrong Fundamental Concept  
of 'Program'?- An Extended Concept", J. Backus,  
in Algorithmic Languages, IFIP, North Holland Pub. Co, 1981.
- [Ber71] "A Computing Machine based on Tree Structures", K.J.Berkling,  
IEEE Trans. Computing., pages 404-418 , C-20, 4, Jan 1971.
- [Ber76] "Reduction Languages for Reduction Machines"  
K.J. Berkling, Proc. Second Int. Symposium on Computer  
Architecture, 1975.
- [Bak80] "Mathematical Theory of Program Correctness"  
Jaco de Bakker, Prentice-Hall International Series in  
Computer Science, 1980.
- [BuPo68] "POP-2 Reference Manual", R. Burstall and R. Popplestone,  
in Machine Intelligence 2, edited by E. Dale and D. Michie,  
Edinburgh Univ. Press.
- [Bur74] "Program Proving as Hand Simulation with a Little Induction"  
R.Burstall, Proc. IFIP Congress 1974, Stockholm.
- [Bus79] "A Dataflow Implementation of Lucid", V.J.Bush,  
M.Sc. Thesis, University of Manchester, 1979.
- [Car67] "Meaning and Necessity, a Study in Semantics and Modal Logic",  
Rudolf Carnap, The University of Chicago Press 1967.
- [Car49] "The Logical Syntax of Languages", R. Carnap,  
International Library for Phil., Psy., and Scientific  
Method 1949
- [Carg76] "Deterministic Operational Semantics for Lucid", Univ. of  
Waterloo, Tech. Rep. CS-76-19, Ontario, 1976.
- [ChKe73] "Model Theory", C. Chang and H. Keisler  
North-Holland Publishing Company 1973.
- [Cla80] "SKIM- The S,K,I reduction machine"  
T.J.W.Clarke et al. Proc. LISP-80 Conference,  
Stanford, Calif. Aug. 1980.
- [Con63] "Design of a separable Transition-diagram Compiler"  
M.E.Conway, CACM, vol 6, No 7, 1963.
- [CuFe58] "Combinatory Logic, Vol I", H.B.Curry and R.Feys,  
North Holland Pub. Co 1958.
- [DaRe81] "ALICE, a Multi-Processor Reduction Machine for the  
Parallel evaluation of Applicative Languages"  
J. Darlington and Mike Reeve. Proc. of the ACM Conf.  
on Functional Programming Languages and Computer  
Architecture, 1981.



- [DML77] "A Highly Parallel Processor Using a Data Flow Machine Language". J.B.Dennis, D.P.Misunas, and C.K.Leung, Computation Structures Group Memo 134, Laboratory for Computer Science, MIT, 1977.
- [Dav78] "The Architecture and System Method of DDM1: a Recursively Structured Data Driven Machine", A.L.Davies Proc. of the 5th Annual Symp on Computer Architecture, Comp. Architecture, 1978.
- [DeMi75] "A Preliminary Architecture for Basic Dataflow Processor", J.B.Dennis and D.P.Misunas, Proc. of 2nd Annual Symposium on Computer Architecture, 1975, pp126-132.
- [Den75] "First Version of a Data Flow Procedure Language" J.B.Dennis, MIT/LCS/TM-61, Cambridge, Massachusetts 1975
- [Den79] " The Varieties of Data Flow Computers " J. Dennis, MIT Laboratory for Computer Science, Computation Structure Group Memo 183, 1979 .
- [Denb81] "A Demand-Driven Coroutine-Based Implementation of a Non-Procedural Language", C.Denbaum, Ph.D. Thesis, University of Iowa, 1983.
- [Far77] "Correct Compilation of a Useful Subset of Lucid", M.Farah, Ph.D. Thesis, University of Waterloo, Ontario 1977.
- [Fau82] "The Equivalence of an Operational Semantics and a Denotational Semantics for Pure Dataflow" A.A.Faustini, Ph.D. Thesis, Theory of Computation Rep. No 41, Dept. of Computer Science, Univ. of Warwick, England.
- [Fin81] "An Operational View of Lucid", B.Finch, Computer Science Department, Report CS-81-37. University of Waterloo, Canada 1981.
- [FMY83] "The P-Lucid Programming Manual", Faustini, Matthews, and Yaghi, Distributed Computing Report 4, University of Warwick
- [FrWi76] "CONS Should Not Evaluate Its Arguments", D.Friedman and D.Wise, in Automata, Languages and Programming, edited by S.Michaelson and R.Milner, Edinburgh University Press 1976.
- [FrWi78] "Unbounded Computational Structures", D.Friedman and D.Wise, Software-Practice and Experience, vol 8, 1978, pp 407-416.

- [Gar78] "Une Implementation de Lucid", P.Gardin,  
Institut D'Informatique, Faculte Universitaires Notre-Dame  
De La Paix Namur, 1978.
- [Gla78] "A Single Assignment Language for Data Flow Computing"  
J.R.W. Glauert, M.Sc Diss., University of Manchester 1978.
- [Gri71] "Compiler Construction for Digital Computers",  
D. Gries, John Wiley and Sons, Inc. 1971.
- [GTW78] "An Initial Algebra Approach to the specification,  
correctness, and Implementation of Abstract Data Types",  
J. Goguen, J. Thatcher, and E. Wagner,  
In Current Trends In programming Methodology, IV,  
Edited by R. Yeh, Prentice Hall Int. 1978
- [GuWa77] " A Multilayered Data Flow Architecture "  
J. Gurd and I. Watson  
Proc. of 1977 International Conf. on Parallel  
Processing , Aug 1976 .
- [Har76] "Truth and Meaning", B. Harrison, in 'Essays in Semantics'  
edited by Gareth Evans and John Mc Dowell  
Oxford University Press 1976.
- [HBS77] "A Universal Modular Actor Formalism", C.Hewitt, P.Bishop,  
and R.Steiger.  
Proc. of Formalism for AI, pages 235-245, 1977.
- [HeBa77] "Actors and Continuous Functionals", C.Hewitt and H.Baker.  
in the Proc. IFIP Working Conf. on Formal Description of  
Programming Concepts held at St.Andrews-Canada, August 1977.  
Ed. E.J.Neuhold, Elsevier North-Holland, N.York 1977,  
Pages 16.1-16.21.
- [HuOp80] "Equations and Rewrite Rules, A survey"  
G.Huet and D.Oppen, SRI Tech. Report CSL-111, Jan 1980.
- [Hen80] "Functional Programming, Application and Implementation",  
P. Henderson, Prentice Hall International Series in  
Computer Science, 1981.
- [HeMo76] "A Lazy Evaluator", P.Henderson and J.Morris,  
Proc. of the 3rd Symp. on Principles of Programming  
Languages, Atlanta, 1976.
- [Hoa69] "An Axiomatic Basis for Computer Programming",  
C.A. Hoare, Comm. ACM , vol 12,10,1969.
- [Hoa78] "Communicating Sequential Processes", C.A.R.Hoare,  
Comm. ACM, Num 8, vol 21, August 1978.
- [HoOd82] "Programming with Equations", C.M.Hoffmann and M.J.O'Donnell,  
ACM Transactions on Programming Languages and Systems,  
vol. 4, No.1, Jan 1982, Pages 83-112.

- [Hof78] "Design and Correctness of a Compiler for a Nonprocedural Language", C.M.Hoffmann, Acta Informatica 9, 1978, Pages 217-241.
- [Hof80] "Semantic Properties of Lucid's Compute Clause and its Compilation", C.M.Hoffmann, Acta Informatica 13, 1980, Pages 9-20.
- [HuLe79] "Computations in nonambiguous linear term rewriting systems", G.Huet and J.J.Levy, IRIA Tech. Report 359, 1979.
- [JaAs84] "Eazyflow: A Hybrid Model for Parallel Processing", R.Jagannathan and E.Ashcroft, Computer Science Lab., SRI International, California 1984.
- [Kah74] "The Semantics of a Simple Language for Parallel Programming" Proc. of IFIP Congress 1974, pp471-475.
- [KaMc77] "Coroutines and Networks of Parallel Processes", G.Kahn and D.McQueen, Proc. of IFIP Congress, 1977, pp993-998.
- [KaMi66] "Properties of a model for parallel computations: Determinancy, termination and queuing", R.M.Karp and R.E.Miller, SIAM Journal of Applied Maths 11,6, 1966.
- [Kel79] "A loosely coupled applicative multiprocessing system", R.M.Keller et al. Proc. of the National Computer Conference, AFIPS Press, Arlington 1978. (pages 861-870).
- [Kow74] "Predicate Logic as Programming Language" R. Kowalski, pages 569-574, Information Processing 74, North-Holland Pub. Co.
- [KLT84] "Rediflow Multiprocessing", R.M.Keller, F.C.Lin, and J.Tanaka. Proc. of IEEE COMPCON, San Francisco, Feb. 1984.
- [Lan64] "The mechanical evaluation of expressions" P.J. Landin, Computer Journal, 6,4, Jan 1964.
- [Lan65] "A correspondence between ALGOL 60 and Church's Lambda-notation", P. Landin, Comm. ACM 8, 1965.
- [Lan66] "The Next 700 Programming Languages", P. Landin, CACM Number 3, vol 9, 1966
- [LuWa69] "On the Formal Description of PL/I", P. Lucas and K. Walk, Ann. Review in Automatic Programming, 6,3, 1969.
- [Man74] "Mathematical Theory of Computation" Z. Manna, McGraw-Hill, Inc. 1974.



- [McC60] "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1", CACM, vol 3, No 4, 1960.
- [Mccl62] "LISP 1.5 Programmer's Manual"  
J. Mc Carthy, et al  
MIT, Cambridge 1962.
- [McC62] "Towards a Mathematical Science of Computation",  
J. Mc Carthy, Information Processing, Proceedings  
of IFIP 1962 (Munich), North Holland.
- [McC65] "Problems in the Theory of Computing", J. McCarthy,  
in the Proc. of IFIP Congress 1965, page 219-22.
- [Mil83] "A Proposal for Standard ML", R.Milner, Univ. of Edinburgh,  
Computer Science Report No. CSR-157-83.
- [Mon74] "Formal Philosophy, Selected Papers of R. Montague",  
Edited by R. Thomason, Yale University press, 1974
- [Nau63] "Revised Report on the Algorithmic Language ALGOL 60"  
P.Naur, et.al., CACM vol 6, Jan., 1963.
- [Odo77] "Computing in Systems Described by Equations"  
Michael J. O'Donnell,  
Lecture Notes in Computer Science 58, Springer-Verlag 1977
- [Ost81] "Luthid 0.0 Preliminary Reference Manual and Report"  
C.B.Ostrum, Computer Science Report, University of Waterloo,  
Ontario, Canada.
- [Pil83] "Translating Lucid Dataflow into Message Passing"  
P.T.Pilgram, Ph.D. Thesis, Dept. of Computer Science  
University of Warwick, Rept. No 5. 1983.
- [Pol72] "Compiler Techniques", Edited by B.Pollack,  
AUERBACH Publishers Inc., 1972.
- [Sar82] "Implementation of Structured Lucid on a Dataflow Computer"  
M.Sc. Thesis, Computer Science Department, University of  
Manchester, 1982.
- [SlBu81] "Towards a zero assignment parallel processor", M.R.Sleep  
and F.W.Burton, in Proc. of the 2nd Int. Conf. Distributed  
Computing, April 1981.
- [Sto77] "Denotational Semantics: The Scott-Strachey Approach  
To Programming Language Theory"  
J.E. Stoy, The MIT Press 1977.
- [Tar56] "Logic, Semantics, Metamathematics", A. Tarski,  
Oxford University Press 1956.
- [TrMo80] "A Multiprocessor Reduction Machine for User-defined  
Reduction Languages", P.C.Treleaven and G.F.Mole,  
Proc. of the 7th International Symp. on Comput. Architecture  
(May 6-8, 1980). IEEE, New York 1980.

- [TBH82] "Data-Driven and Demand-driven Computer Architecture"  
P.C.Treleaven, D.R.Brownbridge, and R.P.Hopkins,  
Computing Surveys, Vol. 14, No. 1, March 1982.
- [Tur81] "The Compilation of an Applicative Language to  
Combinatory Logic", D. Turner, Ph.D. Thesis,  
University of Oxford, 1981
- [Tur82] "Prospects for Non-Procedural and dataflow Languages"  
D.Turner, Proc. Pergamon-Infotech State of the Art Conf.:  
"Programming: New Directions", London 1981.
- [Tur79] "A new Implementation Technique for Applicative Languages"  
D.Turner, Software-Practice and Experience, vol. 9, 1979.
- [WaAs84] "Lucid, The Data Flow Language", W. Wadge and E. Ashcroft.  
Academic Press 1984 (to be published).
- [Wad78] "Away from The Operational view of Computer Science"  
W. Wadge , Theory of Computation Rep. 26,  
University of Warwick, Nov. 1978.
- [Wad83] "Reducibility and Determinateness on the Baire Space"  
W.Wadge, PhD Thesis, Berekley University 1983,  
Also, University of Warwick, Theory of Computation  
Reports 44&45, 1982.
- [Wad82] "Classified Algebras", W.Wadge, CS Report No 46,  
University of Warwick
- [Wad84] "Poppak, A Pascal Implementation of POP-2 Data Types"  
W.Wadge, University of Victoria 1984, Canada.  
"personal communication"
- [Wads71] "Semantics and Pragmatics of the Lambda-calculus"  
C.Wadsworth, Ph.D. Thesis, Oxford University 1971.
- [WaGu82] "A Practical Data Flow Computer", I.Watson and J.Gurd,  
Computer 15(2), Feb, 1982.
- [Wat79] "A Prototype Data Flow Computer with Token Labeling"  
I.Watson and J.Gurd, Proc. of National Computer Conference,  
New York, June 4-7, 1979, vol 48, AFIPS Press.
- [WeEl83] "Strong Typing and Functional Programming- Benefits,  
Design, and Implementation", P.Welch and M.Ellis,  
Computing Laboratory Report, University of Kent, 1983.
- [Yag83] "The Compilation of a Functional Language into Intensional  
Logic", A. Yaghi, Theory of Computing Report, No 56,  
University of Warwick, 1983 .
- [Yag84a] "A Tagged Implementation for Uswim", A.Yaghi,  
University of Warwick Report (in preparation).
- [Yag84] "Higher Order Functions in Lucid", A. Yaghi  
University of Warwick Report (in preparation)